GRANT
IN-61-CR
38735
P51
FINAL

Final Report: Chief and Delta Projects

John Bruner, Hoichi Cheong, David Kuck,
Alexander Veidenbaum, and Pen-Chung Yew (Chief)
Gregory Jaxon, David Padua,
and Paul Petersen (Delta)

December 20, 1990

Project Summary

Title of Effort: Parallel Machine Architecture and Compiler Design Facilities

Grant Number: NCC 2-559

Organization:
Center for Supercomputing Research and Development (CSRD)
University of Illinois
305 Talbot Laboratory
104 S. Wright St
Urbana, IL 61801

Principal Investigators:

David J. Kuck	(217) 333-6150	kuck@csrd.uiuc.edu
Pen-Chung Yew	(217) 244-0045	yew@csrd.uiuc.edu
David Padua	(217) 333-4223	padua@csrd.uiuc.edu
Ahmed Sameh	(217) 333-6352	sameh@csrd.uiuc.edu
Alex Veidenbaum	(217) 244-0043	sasha@csrd.uiuc.edu

Objective:

The objective of the Chief project is to provide an integrated simulation environment for studying and evaluating various issues in designing parallel systems, including machine architectures, parallelizing compiler techniques, and parallel algorithms.

The objective of the Delta project is to provide a facility to allow rapid prototyping of parallelizing compilers that can target toward different machine architectures.

(NASA-CR-188784) PARALLEL MACHINE ARCHITECTURE AND COMPILER DESIGN FACILITIES Final Report (Illinois Univ.) 51 p CSCL 09B

N92-10304

Unclas
G3/61 0038735

Major Accomplishments

1. Developed a program instrumentation and simulation facility, MaxPar, that can measure the maximum inherent parallelism in application programs and also can measure the effectiveness of various parallelizing compiler techniques.
2. Developed parallel simulation kernels on the Alliant FX/8 parallel computer based on a conservative (Chandy-Misra) and optimistic (Time Warp) event-driven models.
3. Developed a parallel simulation kernel, PARSIM, on the Alliant FX/8 parallel computer, that employs a hybrid time- and event-driven model to speed up simulations. PARSIM will also run on uniprocessor machines such as high-performance workstations.
4. Designed and implemented a high-level language CARL (Computer Architecture Research Language), which is based upon C and is used for writing simulators.
5. Developed preprocessors to translate CARL into C and C++ code. The resulting code can be compiled with a standard compiler to allow the simulations to be carried out either on a workstation or on a parallel computer such as the Alliant FX/8.
6. Developed a high-level graphical interface to assist in simulator configuration and to run suites of benchmark executions on the Chief simulators.
7. Developed a bitmapped graphical interface for PARSIM, PARSIM-UI, that allows a user to display and control the state of the simulation. Its operation may be customized with an interpreted language to display simulation-specific information according to user preferences.
8. Developed a data display tool that plots the results of simulation runs on a bitmapped workstation.
9. Implemented two pilot parallel simulators on the Alliant FX/8. They can run a FORTRAN program suite through a parallelizing compiler to generate parallel traces. In one case, the resulting traces drive the simulation of a shared-memory multiprocessor system with a multistage shuffle-exchange network. In the other case, the traces drive the simulation of an eight-processor system similar to an FX/8 system.

1. Chief Project Overview

Chief is a parallel simulation environment for studying parallel systems. Figure 1 shows its basic structure.

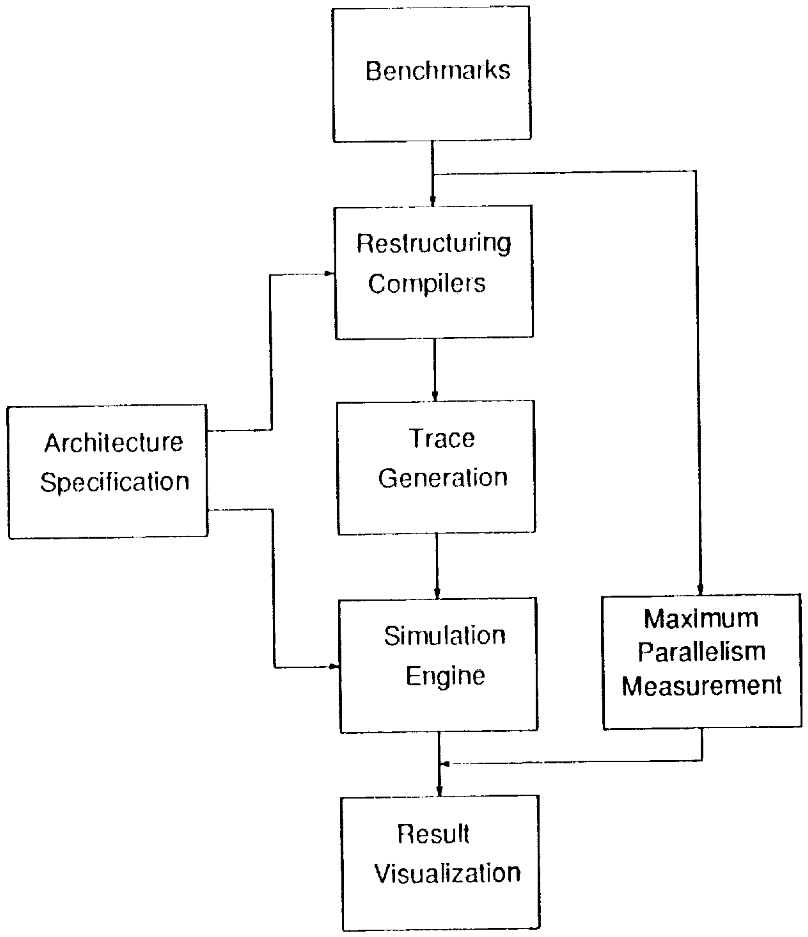

Figure 1 — Chief Project Overview

Parallel systems are studied by creating simulators and driving those simulators with benchmark programs. These benchmark programs are restructured according to the architecture of the target system, and parallel traces are created.

A simulator for the target system is constructed from the architecture specification. The core of the simulator is a simulation kernel (based upon one of three paradigms). The simulator includes a powerful bitmapped window interface that provides the user with a complete view of and control over the execution. The user can vary a set of parameters to the simulated system. The simulator is driven by the parallel traces described above.

Statistics are collected during simulation runs. The Chief environment provides tools to examine these statistics and plot their values against the simulation parameters.

A separate tool, MaxPar, can be used to instrument programs to measure the maximum inherent parallelism within them. The results MaxPar generates are an upper bound on the available parallelism, and can be used to evaluate the effectiveness of the restructuring compilers and simulated system.

2. Simulation Facilities

2.1. Execution-driven Simulation: MaxPar

MaxPar is an execution-driven parallelism profiling and extraction facility. It instruments an application (such as a Perfect Club[1] benchmark) to collect statistics based upon the actual execution of the program. It can determine the inherent maximum parallelism of an application program and the optimal parallelism of the program with system constraints (such as the number of processors, storage-related data dependences, and the synchronization overhead). MaxPar can locate the bottlenecks in the program. Finally, MaxPar can generate parallel execution traces for the program.

MaxPar instruments an application program to record timing and scheduling information for each data object, where a data object is either a scalar variable or an array element. To store this information, MaxPar associates additional variables, called *shadow variables*, with each data object. For each variable X, the read shadow trX records the last time X was read and the write shadow twX records the last time X was written. Given the operation

$$C = A\ op\ B$$

where C, A, and B can be scalar variables or array elements, and the op can be any arithmetic or logical operator, then the equations used to update the shadows are:

$$\begin{aligned} twC &= compute\_time(op) + \max(twC, trC, twA, twB) \\ trA &= \max(trA, twC) \\ trB &= \max(trB, twC) \end{aligned}$$

When a data object is read, its write shadow is checked to determine the earliest possible time for the read operation to proceed. The read can proceed only after the previous write has completed. If the read and write are from different processors, the overhead resulting from data synchronization is computed. The read shadow is then updated to that time. When a data object is written, both its read shadow and its write shadow are checked to compute correct timing and to perform any necessary synchronization.

MaxPar also takes other system features into consideration. The number of processors in the target system may be specified as a finite number or may be infinite. Parallelism may be measured at one of four levels of granularity: operation-level, statement-level, loop-level, or subprogram-level. MaxPar can also take into account scheduling schemes and the synchronization overhead for data synchronization and barrier synchronization. The anti- and output-dependences of a program can be eliminated by an optional dynamic storage allocation scheme. MaxPar can compute the amount of additional storage required to achieve this "pure" data-flow type of execution.

MaxPar instruments the application program, producing a new source program. This is compiled on the host machine, linked with runtime libraries, and executed. The program produces computationally correct answers. In addition, it produces an execution profile by counting the number of operations that can be executed at each time instance. A parallel trace can also be generated. Figure 2 shows the profile of a 512-point fast Fourier transform. The nine "peaks" represent the high parallelism present at the start of each phase of the FFT. The plot does not include the first part of the program, which performs initialization. The parallelism in this example is measured at the loop level with an unlimited number of processors and with no overhead due to scheduling and synchronization.

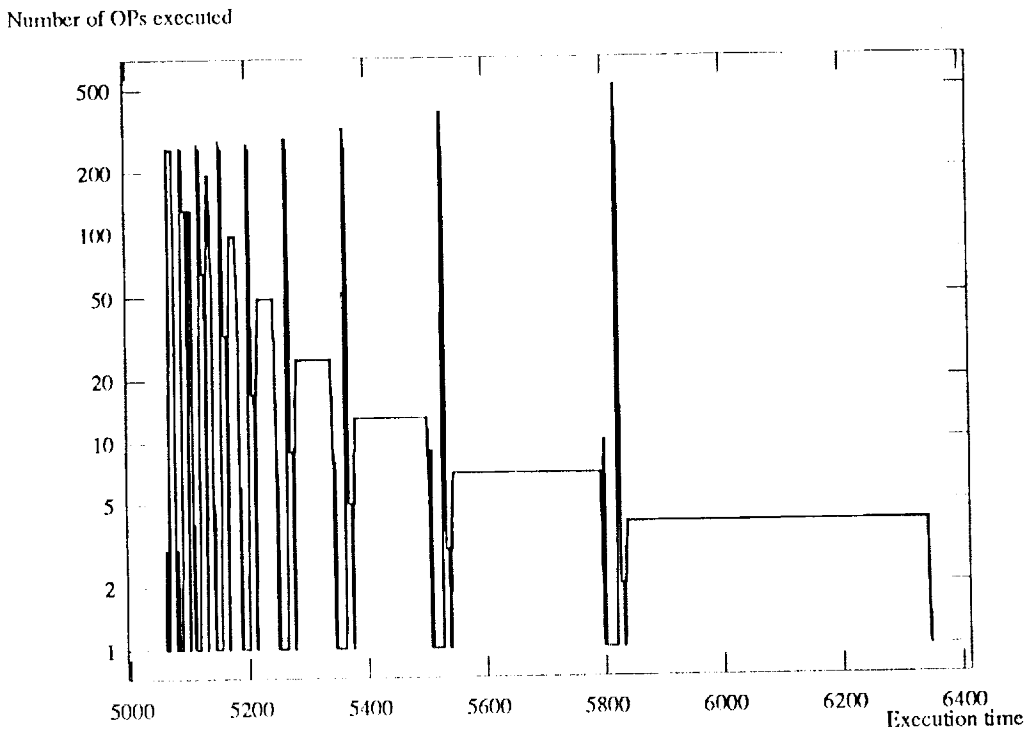

Figure 2 — MaxPar's Execution Profile of a 512-point FFT

2.2. Parallel Simulation Kernels

A Chief simulation consists of a group of modules interconnected by nets. A module encapsulates some function, presenting it to the "outside world" through a set of inputs and outputs. The inputs and outputs of the modules within a system are connected together by nets. The inputs and outputs may be scalars or arrays (with a maximum of three dimensions), the size of which can be specified during runtime configuration. Modules may be implemented directly as a set of low-level functions that directly read values from input nets and write values to output nets. Alternatively, modules may be constructed from other modules. The simulation is implemented as a hierarchy of modules. The root of the module hierarchy is the simulation itself. Many common low-level modules will be provided in a simulation library.

In order to reduce the time required to simulate large parallel systems, Chief provides three different parallel discrete event simulation (PDES) kernels. Simulators built with these kernels share a common user interface, and a single language is used to write code for all three simulation paradigms.

The PDES kernels include a conservative approach (based upon the work of Chandy and Misra[2]), an optimistic approach (based upon the Time Warp[3] technique), and a approach that employs a hybrid of time-driven and event-driven techniques (called PARSIM, for parallel simulator). It is well known that the performance of these PDES approaches is problem- and application-dependent. By providing all three simulation kernels with a single user interface and simulation language, Chief gives users the ability to write one simulator specification and select one of the three approaches at compile time. All three approaches are currently implemented on an Alliant FX/8 system. In addition, PARSIM is also implemented on uniprocessor Sun Microsystems machines.

The user describes each simulation component and the interconnection of components that forms the system. The component definitions are written in the language CARL (described in section 3). Two kinds of components can be defined: behavioral components and hierarchical components. Behavioral components are described by defining their local state, their inputs and ouputs, the actions that should be taken when one or more of their inputs change, and the initialization that should be performed when the simulation starts or is re-executed. Hierarchical components are described by defining the subcomponents that constitute them and the manner in which subcomponents are connected to one another and to the inputs and outputs of the hierarchical component.

A Chief simulator is constructed from a collection of these component definitions. The construction stage comprises two independent phases: the translation phase and the code generation phase. During the translation phase, the component definitions are translated into C structures (for PARSIM) or C++ classes (for Chandy-Misra and Time Warp) that define the various types of the participating components. The data members of each C structure or C++ class represent the state associated with the respective components in the simulation.

For Chandy-Misra and Time Warp simulations, the function members of the C++ class constitute the set of routines needed to simulate the respective components. The system is represented as a collection of logical processes, each of which simulates a component and communicates with other components. Each logical process is the set of member functions defined in its class. An important goal of the construction stage is to to minimize the communication overhead and maximize the potential parallelism in the execution of the simulation. To achieve this goal, we partition the logical processes into sets and assign the simulation of each of these sets to a processor. This assignment is achieved by generating appropriate code to be executed by each physical process.

For PARSIM simulations, the structure definitions are created in a header file and the executable routines that simulate the component are created in a separate code file. The code file is compiled along with the header files of its own component and any included subcomponents to create an executable module. A complete simulation consists of a linked set of of simulation modules.

The execution of the simulation is the final stage of the simulation process. The Chandy-Misra and Time Warp paradigms are based on the exchange of messages to convey information from one component to another. Chandy-Misra also incorporates a means for avoiding deadlock. The machine on which we are developing this tool (an Alliant) is a shared memory machine; therefore, instead of using actual messages we use shared memory to convey information and (in the Chandy-Misra case) to avoid deadlock. By doing so we reduce the cost associated with the use of messages. Each component, for which there are events to simulate is extracted from the ready queue maintained by each physical process, and is simulated on the outstanding events. When there are no more events to simulate it is blocked waiting for new events (messages) to arrive, and control is transferred to another ready logical process. This cycle is repeated until all components have been simulated up to a certain (virtual) time, which has been defined by the user as the `End_of_the_simulation_time`.

PARSIM employs a combination of the time-driven and event-driven approaches to simulation. PARSIM maintains a system event queue that is a time-ordered list of event lists. Each sublist contains events that occur at the same simulation time. PARSIM also maintains event queues for each of the nets affected by clock-induced events.

PARSIM executes events in groups. It dequeues the first list of events from the system event list. Then, in parallel, it evaluates these events, resulting in new values being assigned to nets. Each component that is affected by the change in the nets may specify an "action routine" that updates that components status. PARSIM makes a list of all of the action routines that must be processed. After all of the nets have been updated, all of the action routines are evaluated in parallel. These routines may, in turn, post additional events to the global event queue.

3. CARL — Computer Architecture Research Language

The Chief project provides three different paths by which simulators can be constructed, according to the PARSIM, Chandy-Misra, and Time Warp paradigms, respectively. Although the simulation techniques are different, in all three cases the simulated system is specified as a connected set of hierarchically-defined components.

Components are written in a semi-abstract language called CARL. The use of this language frees the component designer from the need to know the low-level details of the various implementations. More importantly, component definitions written in CARL can be incorporated into any of the Chief simulators simply by using an appropriate preprocessor to convert CARL code to C (for PARSIM) or C++ (for Chandy-Misra and Time Warp).

A component description in CARL consists of sections of C-like code headed by CARL keywords. The keywords are **COMPTYPE**, **INPUTS**, **OUTPUTS**, **SUBCOMPONENTS**, **VAR**, **ACTION**, **INIT**, **STRUCTURE**, **BEGIN**, and **END**. The **COMPTYPE**, **INIT**, **STRUCTURE**, **BEGIN-END** sections contain executable statements modelling a component's behavior and specifying its internal structure.

```
#define ADD   0
#define SUB   1
#define AND   2
#define OR    3

COMPTYPE Alu16(speed)
    int    speed;

INPUTS
    short  in[2]:  alu_eval;
    char   op:     alu_eval;

OUTPUTS
    short  sum;

VAR
    int    Speed;

ACTION alu_eval
    switch (op) {
    case ADD:
        sum = in[0] + in[1] after Speed;
        break;
    case SUB:
        sum = in[0] - in[1] after Speed;
        break;
    case AND:
        sum = in[0] & in[1] after Speed;
        break;
    case OR:
        sum = in[0] | in[1] after Speed;
        break;
    }

BEGIN
    Speed = speed;
END
```

Figure 3 — CARL definition of a 16-bit ALU

Figure 3 shows the CARL definition of a simple ALU, capable of performing four operations upon its two 16-bit inputs. The component, whose type is **Alu16**, has one parameter: the delay between a change to its inputs and a new value on its outputs:

The Chief project includes two preprocessors. PSPP, the PARSIM Preprocessor, converts CARL into C. PSPP is a compiled program that uses the tools "lex" and "yacc" to read CARL programs. It generates two files: a header file that defines PARSIM data structures and a C code file that contains module creation, connection, initialization, and action routines. The host machine's C compiler will convert the code files into object modules that can be linked with the PARSIM runtime libraries and user interface to form a PARSIM simulator.

C2CMTW, the CARL "2" Chandy-Misra/Time Warp preprocessor, converts CARL into C++. C2CMTW, like PSPP, is an executable program. It generates two files: a header file that defines C++ classes for each component type and a C++ code file that contains the definitions of the class member functions. The host machine's C++ compiler will convert these files into object modules that can be linked with either the Chandy-Misra runtime libraries or the Time Warp libraries to create a simulator.

4. UI — PARSIM User Interface

The PARSIM user interface (PARSIM-UI, or simply UI) displays information in bitmapped windows using the X11 window system. It provides control facilities for starting, stopping, continuing, and breakpointing simulation runs. Nets can be viewed graphically. By creating several windows, the user can interact with the simulation from multiple contexts.

The core of PARSIM-UI is an execution engine that parses and executes commands written in a simple language. The graphical interface "wrapper" accepts input in the form of menu selections, button presses, etc. and transforms it into commands that are interpreted by the engine. The user-interface language is also directly available, so that the user can customize his or her debugging sessions as necessary.

PARSIM-UI can directly access objects in the simulation system: components, inputs, outputs, and nets. It also provides and operates upon simulator variables. Variables may contain integer, floating-point, or string values, or may contain one of three special typed values: error, high-impedance, and unknown. Their type is dynamic — an assignment to the variable sets the type as well as the value. The value of an uninitialized variable is the integer zero.

A set of operators combines components, nets, variables, and literal constants into more complex expressions. An expression may be used whenever the PARSIM user interface expects a value. In particular, an expression may be used within a component or net array subscript. Function calls may also appear within expressions. They are called using the syntax

function_name (*args*)

where *function_name* is the function name and *args* is a comma-separated list of expressions that represent the arguments to the function. The number and type of arguments are function-specific. PARSIM-UI provides a set of standard built-in functions, which provide access to the simulation state. Users can define additional functions.

The primary interface to PARSIM-UI is graphical; however, in recognition of the fact that text input is sometimes necessary, macros can be used to hid some of the programming-language appearance from the user. A set of built-in macros is provided. The user may define any number of new macros and is free to redefine the built-in macros if he or she so desires.

The PARSIM-UI language provides primitives for grouping, iteration (**WHILE**), conditionals (**IF**), function definition, and macro definition. The syntax is vaguely similar to Algol or C.

PARSIM-UI provides a powerful breakpoint facility. Breakpoint conditions are expressed as an arithmetic expression and therefore may depend upon nets, constants, and variables. This provides great flexibility; for instance, it is possible to check if the currently-addressed register in a register file is zero or if the values of two registers are equal. When a breakpoint condition is satisfied a second expression (the breakpoint ‘‘action’’) is evaluated. The action may include stopping simulation after the current simulation time step, but it need not do so. Other possible actions might include printing a message, updating a display, or collecting statistics in a counter.

The user may display any subset of the simulation state by defining one or more *autodisplays*. An autodisplay is a window that continuously displays sets of expressions. Effectively it is a snapshot of a user-specified subset of the current simulation state. Autodisplays allow the user to create views of collections of nets and to watch them change as the simulator executes. They are updated each time that the simulator stops (*e.g.*, due to a breakpoint) and at other times as directed by the user or breakpoint definitions. The appearance of an autodisplay window is primarily under the user’s control.

The contents, format, and location of all autodisplay items are user-configurable. The fields within an autodisplay can be moved and resized using the mouse.

PARSIM-UI can save any part of its user interface state to a file. Thus, the definition of one or more autodisplays, breakpoints, functions, macros, and/or variables can be preserved from one simulation run to the next. This gives the PARSIM-UI user significant control over the configuration of his or her environment and makes PARSIM-UI a powerful tool for debugging and running simulations.

5. Parallel Trace Generation Facilities

5.1. Optimal Parallel Traces (MaxPar)

MaxPar can produce an optimal parallel trace by instrumenting a program at the source code level with tracing instructions. The traces that are generated when the program is executed are optimal in the sense that they reflect the best possible parallelism within a program; therefore, they can provide an idealistic performance upper bound for the program. The information in this trace is independent of the machine architecture and the parallelizing compiler.

5.2. Symbolic Parallel Traces (Parafrase)

Symbolic parallel traces are used for generic shared-memory systems with an optional vector processing unit in each processor. The set of traces generated can be targeted toward a particular machine organization such as SIMD, MIMD, etc. It provides users with a good mix of realistic computer architecture characteristics, and also allows them to specify particular characteristics of their own machines. For example, a user can specify the number of processors in the system, the scheduling scheme, the data layout in the shared memory, etc. The resulting parallel traces reflect the possible parallelism that can be obtained by a parallelizing compiler as opposed to the maximum parallelism that can be obtained using MaxPar.

Parafrase-based parallel trace generation consists of three steps:

1. *Generate program intermediate form.* The syntax and the semantics of the intermediate language resemble an assembly language for a vector multiprocessor. The output is generated for each subroutine separately. An infinite number of symbolic registers is assumed for the system.
2. *Link and load modules for execution.* This resolves symbolic references, processes parameter passing, lays out common blocks, determines which data is global and which is local, and produces a load map.
3. *Execute the load modules symbolically.* This step simulates parallel execution of the load modules and produces time stamps for instructions accessing memory. The result is a

memory reference trace that can be used to drive multiprocessor simulators created by the Chief tools.

5.3. Alliant FX/8 Traces

Alliant supplies an emulator for the FX/8. Programs are compiled with the parallel Alliant Fortran or C compiler. The resulting object modules are linked with runtime routines to create executables. These are then emulated to produce Alliant-specific memory reference traces. The programs produce computationally correct results, and the traces are a very realistic reflection of the program's parallel behavior. However, because the traces are machine-specific, they cannot be made to accomodate a memory hierarchy or a configuration consisting of more than eight processors.

6. Data Visualization Tool

The Chief visualization tool plots data for display on a bitmapped workstation. The data is collected from a suite of simulation runs in which simulation parameters are varied from run to run. The data from each run is stored into a file. A separate description file identifies all of the data items. The visualization tool reads the description file and all of the data files. The user can plot any data item against any simulation parameter while constraining the values of other simulation parameters.

7. Top-Level Chief Environment

All of the Chief tools are assembled into a top-level bitmapped environment. The environment guides the user through the creation of a simulator from a set of components stored in a component library. More than one version of some components may be archived, so the environment allows the user to view the current set of components and select the desired version for each one. Each component contains a set of parameters that control its behavior. The environment extracts a complete list of parameters from the specified components and provides mouse-driven tools that allow the user to specify new parameter values.

The environment provides a simple interface that allows the user to specify a set of compiler parameters, compile a benchmark, and generate a trace file. The editing of compiler parameters is similar to the editing of simulation parameters. In addition, the environment also allows the user to invoke MaxPar to analyze the parallelism within the benchmark.

When instructed to build a simulator, the environment will invoke the appropriate Chief preprocessor for each component definition (written in CARL), will invoke the system compiler to create object files for all components, and will link those object files with the appropriate kernel and user interface libraries. A simple command will execute the resulting simulator.

The power of the Chief environment lies in its ability to execute a suite of compilation and simulation runs while varying the input parameters. The user specifies a set of values for each parameter, and the environment will automatically compile the benchmark to produce a trace file, build the simulator, and invoke the simulator with the trace file as input. The output from each simulation run will be written to a separate file. The user can then use the Chief visualization tool described above to display these results graphically.

8. Pilot Simulations

Two simulators have been developed to demonstrate the utility of the Chief environment. First, the simulations can be run in parallel, resulting in fast execution time. Furthermore, the simulations are written in CARL, which is extremely modular, allowing faster initial code development. This allows component models to be replaced much more easily than in dedicated simulation programs.

8.1. Cedar-like System Simulator

A simulator has been developed to simulate the Cedar global memory system[4,5]. It consists of models for the Omega networks, the global memories, and a simple processor. The simulator is driven by traces of Fortran programs generated by Parafrase. It allows different system configurations to be simulated by changing the size of the system and the size and configuration of the network switches, and by replacing the switch and memory component models to test, *e.g.*, different internal buffering configurations.

The system model is a simplification of Cedar, in that processors are not clustered as in Cedar. Furthermore, the current processor model does not simulate the effects of caching or cluster memory. Some of these effects can be accommodated by changing the way traces are gathered by Parafrase. More advanced processor models are also under development.

8.2. Cedar Cluster Simulator

A simulator has been developed to simulate a Cedar cluster. It consists of models for the caches, the cluster memory, and eight simple processors. Traces created by the Alliant FX/8 emulator drive the simulation.

References

1. M. Berry, *et. al.*, "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers," *The International Journal of Supercomputer Applications*, vol. 3, pp. 5-40, 1989.
2. K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, vol. 24, no. 4, pp. 198-206, April 1981.
3. D. R. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, pp. 404-425, July 1985.
4. P. Yew, "The Architecture of the Cedar Parallel Supercomputer," *Parallel Systems and Computation*, edited by G. Almasi, R. Hockney, and G. Paul (North-Holland, Amsterdam), pp. 137-148, 1988.
5. D. J. Kuck, E. Davidson, D. Lawrie, and A. Sameh, "Parallel Supercomputing Today and the Cedar Approach," *Science*, pp. 967-974, February 1986.

NASA NCC 2-559

[Amma90] Zahira Ammarguellat. *A Control-Flow Normalization Algorithm and its Complexity.* **Submitted for publication**, June 1990.

[Andr90] John Barrett Andrews. "A Hardware Tracing Facility for a Multiprocessing Supercomputer", CSRD Report No. 1009, UILU-ENG-90-8026, Univ. of Illinois at Urbana-Champaign, Ctr. for Supercomputing Res. & Dev., May 1990.

[BCVY90] John Bruner, Hoichi Cheong, Alexander Veidenbaum and Pen-Chung Yew. *Chief: A Parallel Simulation Environment for Parallel Systems.* **Submitted for publication**, November 1990.

[Berr90] Michael Waitsel Berry. "Multiprocessor Sparse SVD Algorithms and Applications", CSRD Report No. 1049 UILU-ENG-90-8031, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., November 1990.

[Brun90] John Bruner. "PARSIM User Interface Reference Manual", CSRD Report No. 1002, Univ. of Illinois at Urbana-Champaign, Ctr. for Supercomputing Res. & Dev., September 1990.

[Brun90] John Bruner. "PARSIM User Interface Tutorial", CSRD Report No. 1040, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., September 1990.

[ChHa90] Jyh-Herng Chow and Luddy Harrison. *Switch-Stacks: A Scheme for Microtasking Nested Parallel Loops.* **Proc. of Supercomputing '90, New York, NY**, pp. 190-199, November 12-16, 1990.

[ChVe90] Hoichi Cheong and Alexander Veidenbaum. *Compiler-directed Cache Management in Multiprocessors.* **IEEE Computer Journal**, Vol. 23, No. 6, pp. 39-47, June 1990.

[ChVe90] Yung-chin Chen and Alexander V. Veidenbaum. *Comparison and Analysis of Software and Directory.* **Submitted for publication**, November 1990.

[DaYe90] Timothy Davis and Pen-Chung Yew. *A Stable Non-deterministic Parallel Algorithm for General Unsymmetric Sparse LU Factorization.* **SIAM J. on Matrix Analysis and Applications**, Vol. 2, No. 3, pp. 383-403, July 1990.

[FuCh90] Chuigang Fu. "Evaluating the Effectiveness of Fortran Vectorizers by

Measuring Total Parallelism", CSRD Report No. 1033, UILU-ENG-90-8029, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., August 1990.

[GaBe90] Hui Gao and Michael Berry. "Performance Studies of LAPACK on Alliant FX/80 and 1 Cedar Cluster", CSRD Report No. 1001, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., May 1990.

[GoGV90] Edward H. Gornish, Elana D. Granston and Alexander V. Veidenbaum. *Compiler-directed Data Prefetching in Multiprocessors with Memory Hierarchies.* **Proc. of ICS'90, Amsterdam, The Netherlands**, Vol. 1, pp. 354-368, May 1990.

[GrVe90] Elana D. Granston and Alexander V. Veidenbaum. *Detecting Opportunities for Data Reuse.* **Submitted for Publication**, November 1990.

[HsYe90] William Tsun-yuk Hsu and Pen-Chung Yew. *An Effective Synchronization Network for Large Multiprocessor Systems.* **Subm. the 5th Int'l Parallel Processing Symp., Disneyland Hotel Convention Center, April 30 – May 2, 1991**, September 1990.

[LiYe90] David J. Lilja and Pen-Chung Yew. "The Interaction of Cache Block Size and Parallel Loop Scheduling Strategy", CSRD Report No. 989, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., July 1990.

[LiYe90] David Lilja and Pen-Chung Yew. *The Performance Potential of Fine-Grain and Coarse-Grain Parallel Architecture.* **Submitted for publication**, June 1990.

[LiYe90] David J. Lilja and Pen-Chung Yew. "A Compiler-Assisted Directory-Based Cache Coherence Scheme", Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., November 1990.

[LiYe90] David J. Lilja and Pen-Chung Yew. *Combining Hardware and Software Cache Coherence Strategies.* **Submitted for publication**, December 1990.

[Malo90] Allen Davis Malony. "Performance Observability", CSRD Report No. 1034, UILU-ENG-90-8030, Univ. of Illinois at Urbana-Champaign, Cntr. for Supercomputing Res. & Dev., September 1990.

[MaLR90] Allen Malony, John Larson and Daniel Reed. *Tracing Application Program Execution on the Cray X-MP and Cray 2.* **Proc. of Supercomputing**

'90, New York, NY, pp. 60–73, November 12–16, 1990.

[MiPa90] Sam Midkiff and David Padua. *Issues in the Compile-Time Optimization of Parallel Programs.* **Proc. of Int'l Conf. on Parallel Processing 1990**, Vol. II, pp. 105–113, August 1990.

[Padu89] David Padua. *Problem Solving Environments for Parallel Computing.* **Proc. of an International Conf. organized by the IPSJ to Commemorate the 30 Anniversary**, pp. 323–330, November 1990.

[Poin90] Lynn Pointer. "Perfect: Performance Evaluation for Cost–Effective Transformations Report 2", CSRD Report No. 964, University of Illinois at Urbana–Champaign, Center for Supercomputing Res & Dev, March 1990.

[PoJa90] David Pointer and Greg Jaxon. "Cedar Synchronization Processor Instruction Set Reference", CSRD Report No. 1017, Univ. of Illinois at Urbana–Champaign, Center for Supercomputing Res & Dev., July 1990.

[Shar90] Sanjay Sharma. *Real–Time Visualization of Concurrent Processors.* **To be presented at the Joint Conf. on Vector and Parallel Processing, Zurich Switzerland**, November 1990.

[ShLY90] Zhiyu Shen, Zhiyuan Li and Pen–Chung Yew. *An Empirical Study of Fortran Programs for Parallelizing Compilers.* **IEEE Trans. on Parallel and Distributed Systems**, pp. 350–364, July 1990.

[ShSh90] Priyamvada Sinvhal–Sharma and Sanjay Sharma. "CPROF: A Trace Based Profiler for Shared Memory Multiprocessor Systems", CSRD Report No. 1016, Univ. of Illionis at Urbana–Champaign, Cntr. for Supercomputing Res. & Dev., June 1990.

[SMBS90] Sanjay Sharma, Allen Malony, Michael Berry and Priyanvada Sinvhal–Sharma. *Run–Time Monitoring and Performance Visualization of Concurrent Programs.* **Proc. of Supercomputing 90, November 1990, New York, NY**, pp. 784–793, November 12–16, 1990.

[YeBr90] Pen–Chung Yew and John Bruner. *SEE: A System Evaluation Environment for Studying Parallel Systems.* **To be presented at the First Workshop on Parallel Processing, National Tsing Hua, Univ., Taiwan, Dec. 20–21, 1990**, December 1990.

[Chen89] Ding–Kai Chen. "MaxPar: An Execution Driven Simulator for Studying Parallel Systems", CSRD Report No. 917, UILU–ENG–89–8013, Univ. of Illinois at Urbana–Champaign, Center for Supercomputing Res. & Dev., October 1989.

[Cheo89] Hoichi Cheong. "Compiler-Directed Cache Coherence Strategies for Large-Scale Shared-Memory Multiprocessor Systems", CSRD Report No. 953, UILU-ENG-89-8018, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., December 1989.

[ChSY89] Ding-Kai Chen, Hong-Men Su and Pen-Chung Yew. *The Impact of Synchronization and Granularity on Parallel Systems.* **To appear in the Proc. of the 17th Int'l. Symp. on Computer Architecture, Seattle, WA**, December 1989.

[Davi89] Timothy Alden Davis. "A Parallel Algorithm for Sparse Unsymmetric Factorization", CSRD Report No. 907, UILU-ENG-89-8012, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., September 1989.

[GaFM89] E. Gallopoulos, G. Frank and U. Meier. *Experiments with Elliptic Problem Solvers on the Cedar Multicluster.* **Proc. of Fourth SIAM Conf. Par. Proc. Sci. Comput., Chicago, IL**, pp. 245-250, December 1989.

[GaSa89] E. Gallopoulos and Youcef Saad. *Some Fast Elliptic Solvers on Parallel Architectures and Their Complexities.* **Int'l. J. High Speed Computing**, Vol. 1, No. 1, pp. 113-141, May 1989.

[Gorn89] Edward H. Gornish. "Compile Time Analysis for Data Prefetching", CSRD Report No. 949, UILU-ENG-89-8016, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., December 1989.

[Koss89] Peter Koss. "Application Performance on Supercomputers", CSRD Report No. 847, UILU-ENG-89-8001, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., January 1989.

[Lave89] Daniel M. Lavery. "The Design of a Hardware Performance Monitor for the Cedar Supercomputer", CSRD Report No. 866, UILU-ENG-89-8006, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., May 1989.

[Lilj89] David Lilja. *Efficient Generation of Poisson Distributed Random Numbers.* **Trans. of Soc. for Comp. Simulation**, Vol. 6, No. 1, pp. 31-41, 1989.

[LiMY89] David Lilja, David Marcovitz and Pen-Chung Yew. "Memory Referencing Behavior and a Cache Performance Metric in a Shared Memory Multiprocessor", CSRD Report No. 836, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., February 1989.

[LiYZ89] Zhiyuan Li, Pen-Chung Yew and Chuan-Qi Zhu. *Data Dependence Analysis on Multi-Dimensional Array References.* **Proc. of 1989 Int'l. Conf. on Supercomputing**, pp. 215-224, June 1989.

[LiZh89] Zhiyuan Li. "Intraprocedural and Interprocedural Data Dependence Analysis for Parallel Computing", CSRD Report No. 910, UILU-ENG-89-8011, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., August 1989.

[MiPC89] Samuel P. Midkiff, David Padua and Ronald G. Cytron. *Compiling Programs with User Parallelism.* **Research Monographs in Parallel & Distributed Computing, C. Jesshope, D. Klappholz (eds.), Pitman Publishing, 1989.**, 1989.

[Suzu89] Hiroshi Suzuki. "A Serial Communication Interface for a Parallel Simulation System", CSRD Report No. 950, UILU-ENG-89-8017, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., December 1989.

[TaYZ90] Peiyi Tang, Pen-Chung Yew and Chuan-Qi Zhu. *A Parallel Linked List for Shared-Memory Multiprocessors.* **Proc. of the 13th Annual Int'l Computer Software & Application Conf.**, pp. 130-135, September 1989.

[Turn89] Stephen Wilson Turner. "Shared Memory and Interconnection Network Performance for Vector Multiprocessors", CSRD Report No. 876, UILU-ENG-89-8007, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., May 1989.

[TuVe89] Stephen Turner and Alexander Veidenbaum. "Burst Traffic in MIN-Based Shared Memory Systems", CSRD Report No. 855, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., February 1989.

FINAL

Project Summary
The Delta Program Manipulation System *

Gregory Jaxon David Padua Paul Petersen
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign

March 1, 1991

Abstract

This report summarizes the status of the Delta Program Manipulation System [Pad89] project at the expiration of its initial project development grant. Included are a review of the project's objectives and surveys of the program manipulation tools developed, the environmental software supporting Delta, and the compiler research projects in which Delta has played a role. An appendix describes the Delta system in detail.

1 Objectives

FORTRAN 77 programs are portable to many computer architectures. But the program characteristics that yield the best performance vary from machine to machine. The common goal of researchers in automatic restructuring is to capture and preserve the meaning of a program while varying the program structures that most influence its speed and efficiency on different computer systems. Although a number of commercial and research program restructurers have been written, the cost of exploring new techniques or optimization strategies is still extremely high.

The Delta Program Manipulation System[Pad89] is an open system of tools and components and a workbench environment for developing new compiler techniques in automatic program restructuring. Included are: a FORTRAN parser; an extensive repertoire of operations and data structures common to vectorizing and parallelizing compilers; and the tools and methodology needed to generate and test new compilation methods and strategies. We believe that this approach can reduce the cost of research and development for advanced compilers in the same way that domain-specific languages (e.g. Mathematica) have reduced the cost of problem solving in other technical fields.

Openness An 'open system' is one which exposes its component parts for modification, replacement, or reuse in new contexts. Several factors contribute to the openness achieved in Delta.

*This work was supported by the National Aeronautics and Space Administration and the Defense Advanced Research Projects Administration under Grant No. NASA NCC 2-559. Part of this work was carried out by James R.B. Davies.

CASI

- The implementation language (SETL) is very high level. This means the amount of text invested in any one design commitment tends to be small, and thus manageable.
- An 'applicative' programming methodology has been followed in which components are relatively insensitive to the context in which they are used.
- The central data structures in Delta are labelled maps. Because they are self-documenting and flexible, they are easy to use in new ways, or modify for new uses.
- Environmental software has made the SETL source of Delta 'content addressable'.

In the following sections we will describe parallelization and illustrate Delta's program manipulation tools, then survey the environmental software supporting Delta and the compiler technology research at CSRD in which Delta is beginning to play a part. Appendix A gives a detailed description of the Delta system.

2 Programs and Manipulations

Delta operates on FORTRAN77 programs. To make them tractable, they are represented internally as abstract syntax trees that supress the lexical and syntactic quirks of FORTRAN. In SETL, data objects can share storage under a discipline of 'copy-on-write'. Each FORTRAN program appears to be a separate SETL data object. Delta transformations take FORTRAN programs as 'call-by-value' arguments and deliver revised programs as results. Memory requirements do not multiply since only the few substructures which change need new storage.

Internally, Delta breaks a program into its:

- Symbol table
- I/O format specifiers
- Imperative statements
- Applicative expressions
- Storage equivalences
- Initial data values
- Common storage layout

Each substructure collects and indexes one class of program components. The component descriptions are collections of named attributes. Some attributes link components together (by their names or indices) into semantic networks. Delta works by discovering and deriving facts about the program's behavior when it is executed. Facts are added to the tree both as new top level structure and as annotations to low level components.

An executing FORTRAN program produces a *sequence* of stores into memory cells, references to stored values, and calculations creating new values. The program's text may refer to one storage cell in many different ways. The cells of an array are identified by subscripts which are integer arithmetic formulae. Symbolic algebra and Diophantine analysis can be used to test whether two subscript formulae ever intersect. Where they do, the two uses of that array may involve the same storage cell. Such a pair, where at least one storage action is a write, forms a **data dependence** and requires that the two memory references occur in their original order.

The sequencing of storage actions is captured in data dependence graphs, a control flow graph, and a subroutine call graph. The graphs summarize how the parts of the program cooperate to achieve its net result. These graphs are examined before most program changes to verify that the transformed code will be equivalent to the original. As the program is changed, these graphs are updated or regenerated to reflect the current organization of the program. The incremental cost to do this is small because optimizing transformations tend to *preserve* most storage relationships.

Parallelism can be recognized in a sequential program as a pattern of data and control *in*dependence. Parallelizing is the process of producing these patterns by modifying loop structure, introducing auxiliary storage cells, and reorganizing calculations to avoid small cycles of dependence which can only be supported by serial loops.

Today the Delta system includes sufficient preconditioning, analysis, and transformation components to parallelize and restructure many example programs. It can permute the nesting order of a collection of loops, distribute loops into vector form, or split them into parallel and serial pieces. It can normalize them, stripmine them, or reverse their iteration spaces. It recognizes scalar inductions carried by a single loop, scalar variables local to a single loop, summations, and DOALLs.

In the next half year we will extend Delta's parallelization techniques by collecting dependence cycle breakers: particular transforms, triggered by the appearance of a circular path in the data or control dependence graph, and designed to break the cycle. Some of these cycles are easily broken by recognizing which variables are loop invariant, linearly varying, or localizable. Such properties of a loop are discovered by the preconditioning passes already built for Delta and appear as annotations to the internal program representation for later passes to use.

3 Environmental Support

3.1 SETL

For now, any serious user of Delta must become a SETL programmer. Fortunately most programmers can intuit the basic principles of SETL by imagining a cross between Algol control constructs, Set Theory notation, and Lisp recursive data structures. A key to the power of SETL is the flexibility of sets and tuples for representing data structure. It is especially important for Delta programmers to understand maps. A map is a set of ordered pairs (i.e. 2-tuples). SETL allows a map to be applied to an argument like a function; the result is the second element of the ordered pair whose first element matches the argument. For example, if we create a map from the first four integers to their names:

```
>     number_to_name := {[1,"one"], [2,"two"], [3,"three"], [4,"four"]};
```

then we can use this variable like a function:

```
>     number_to_name(1);
"one";
```

If the argument is not in the domain of the map (i.e. the set of first elements of the ordered pairs), the mapping operation returns 'OM'. If more than one ordered pair has the same first element, then the map is referred to as multi-valued. A special form of the mapping operation, using curly braces instead of parentheses, will return the set of all second elements of ordered pairs in the map whose first element matches the argument:

```
>    number_to_name := number_to_name union {[1,"uno"]};
>    number_to_name{1};
{"uno", "one"};
>    number_to_name{2};
{"two"};
```

A mapping operation with parentheses is illegal for members of the domain with multiple values. The test for this error occurs at runtime. The choice of which algorithm to use to perform the mapping is also made at runtime. Very little of SETL's syntax is devoted to specifying implementation details. Runtime choices are expensive. They are avoidable in a commercial restructurer, but are welcome in Delta because they reduce the amount of text that must be changed to revise a design choice.

3.2 Interactive Delta

A typical Delta development session might start out as follows. First the interpreter is started and it reads all of the Delta source code:

```
shell% idelta
DELTA Program Manipulator   Last update: Feb 14 16:09
(c) 1991, Board of Trustees, Univ. of Illinois (CSRD)
ISETL 2.0  Last updated on 89/12/12 at 13:18:09.
(c) Copyright 1987,1988,1989 Gary Levin
Enter !quit to exit.

Current GC memory = 50080, New Limit = 4000000
Current GC memory = 3996384, Limit = 4000000
>       matmul := read_program("matmul.f");
>       display_program(matmul);
            SUBROUTINE MATMUL(A,B,C,N)
   S2       DO I = 1, N, 1
   S3        DO J = 1, N, 1
   S4         X = 0.0
   S5         DO K = 1, N, 1
   S7          X = X+B(I,K)*C(K,J)
   S8         ENDDO
  S10         A(I,J) = X
  S11        ENDDO
  S12       ENDDO
  S13       RETURN
            END
OM;
```

Function read_program invokes the Delta scanner (a separate program, written in C) on its filename argument. The scanner produces a SETL data structure that completely describes the program. Read_program loads this structure as a variable within the ISETL session, annotates it with its control flow graph and variable cross reference, and returns the whole package as its functional result. Here we assigned it to the variable 'matmul'. A call to display_program lists out the executable statements of 'matmul' in FORTRAN form.

The '>' is an iSETL prompt; a statement typed here will execute and have its value printed. The value of display_program was undefined, which SETL treats as a constant called 'OM' (for Omega or omitted). We can begin to examine matmul as a SETL map by asking:

```
>       domain(matmul);
{"statements", "initial_statement", "final_statement",
 "expression", "loop_info", "routine_type", "symtab"};
```

In more complex programs we might also see substructures for "common_blocks", "equivalences", and other FORTRAN features.

Compiler authors rely on utility functions to abbreviate most data accesses. For example, one query function in Delta is called stmts_of_type. It returns a tuple of names of statements of a given type, in the order that they appear in the program. If lexical order is not important, the raw SETL needed to acquire the same subset is almost as brief:

```
>       stmts_of_type(matmul,"DO");
["S2", "S3", "S5"];
>       all_stmts := matmul("statements");
>       { stmt : attr = all_stmts(stmt) | attr("st") = "DO" };
{"S2", "S5", "S3"};
```

At the top level of compiler construction, transformation and analysis functions are more common. Here we have composed many Delta steps into a 'precondition' function, which returns a heavily annotated version of its argument. We then apply an experimental vectorizer to the annotated program and put the parallelized result into a separate iSETL variable called 'matmul_vector'.

```
>       matmul_vector := tiny_vectorizer (precondition (matmul));
S2 is a DOALL
S3 is a DOALL
S5 is a summation
```

We have chosen to express parallelism as an annotation to a fundamentally serial program. Preserving the sequential view of the program's semantics means that sequential analyses are still applicable to the transformed program.

```
>       display_program (matmul_vector);
            SUBROUTINE MATMUL(A,B,C,N)
   S2       DO I = 1, N, 1  {DOALL}
   S3        DO J = 1, N, 1  {DOALL}
   S4         X = 0.0
   S5         DO K = 1, N, 1  {SUM}
   S7          X = X+B(I,K)*C(K,J)
   S8         ENDDO
  S10         A(I,J) = X
  S11        ENDDO
  S12       ENDDO
  S13       RETURN
            END
OM;
>       print_graph(matmul_vector, "S5", OM);
Output Dependence with Direction [=, =, <] from S7[26]: X  to S7[26]: X
  Flow Dependence with Direction [=, =, <] from S7[26]: X  to S7[27]: X
   Antidependence with Direction [=, =, <] from S7[27]: X  to S7[26]: X
OM;
```

On the other hand, most restructuring transformations also modify the serial program. Changes to the lexical and control flow graphs are evident in the program display.

```
>       display_program (stripmine (matmul_vector, "S3", 32));
S3 stripmined into [S3, ST2]
            SUBROUTINE MATMUL(A,B,C,N)
   S2       DO I = 1, N, 1  {DOALL}
   S3        DO J1 = 1, N, 32  {DOALL}
  ST2         DO J = J1, MIN(N,31+J1), 1  {DOALL}
   S4          X = 0.0
   S5          DO K = 1, N, 1  {SUM}
   S7           X = X+B(I,K)*C(K,J)
   S8          ENDDO
  S10          A(I,J) = X
  ST3         ENDDO
  S11        ENDDO
  S12       ENDDO
  S13       RETURN
            END
OM;
```

An interactive Delta session allows a compiler developer to explore potential algorithms by !include'ing experimental software and watching its effects on actual programs. The growing repertoire of transformers, instrumenters, program analyses and displays make the interactive system a versatile laboratory for developing new compiler technology.

Using iSETL within Delta

Delta uses maps (and occasionally multi-valued maps) where other compilers would use structured data. For instance, each FORTRAN compilation unit is a map. When we ask for `matmul("statements")`, the result is a map from statement names onto other maps that describe the attributes of each statement. For example the right hand side of the assignment in statement S7 is:

```
>       format_expr(matmul, matmul("statements")("S7")("rhs"), true);
"X+B(I,K)*C(K,J)";
```

Another statement attribute is its FORTRAN type (spelled `"st"`). To explore the concept of SETL maps, let's quickly build, and then display, a map from statement names directly to their type:

```
>    stmt_types:= {[s, attr("st")] : [s,attr] in matmul("statements")};
>    stmt_types("S1");
"DO";
```

Here s and attr are iteration variables; they become bound to the elements of each ordered pair in the statement map of matmul. Attr is the map of the attributes of statement s; attr("st") is the one we we need. As s and attr iterate over all the statements, {[s_1, $attr_1$("st")], ...} becomes a set of ordered pairs: a new mapping.

SETL can iterate over maps, sets, or tuples to form other maps, sets, or tuples, or to do more traditional forms of data processing. In the following (truncated) example, the '>>' prompt indicates that iSETL is waiting for more text in a syntactically incomplete construct (in this case a conventional 'for' loop).

```
>    for t=stmt_types(s) do
>>      writeln s," is ", if t(1) in "AEIOU" then "an "else "a "end if, t;
>>   end for;
S12 is an ENDDO
S13 is a RETURN
S4 is an ASSIGNMENT
S2 is a DO
S1 is an ENTRY
        ⋮
```

A SETL Extension: @fieldname

Since we make such extensive use of strings in the domain of maps, we have extended iSETL to streamline the syntax for mapping strings. The syntax resembles function composition: @xxx@yyy[z] finds the location "xxx" in the map called "yyy" in the map stored in variable z. This notation may be used for both storage to and retrieval from a hierarchical dataset.

```
>    @st@S4@statements[matmul];
"ASSIGNMENT";
>    stmt_types:= {[s, @st[attr]] : [s,attr] in @statements[matmul]};
>    @S2[stmt_types];
"DO";
```

3.3 Batch Processed Delta

iSETL is not the fastest implementation of SETL. Production runs of Delta on large programs are not practical using the public domain interpreter that makes development so easy. To overcome this problem, We acquired the SETL2 compiler from Courant Institute[Sny90] for our workstations. In many cases, the compiled form of Delta has proven to be between 4 and 20 times faster than interactive iSETL interpretation. This enables us to process the entire Perfect Benchmark suite through a Delta experiment as an overnight batch job.

iSETL → SETL2 conversion

The SETL2 compiler and iSETL interpreter differ in many interesting respects. We have avoided substantial parts of both languages in an attempt to keep a single source code for Delta in a fairly standard core dialect. The remaining differences are bridged by writing in a subset of iSETL and converting iSETL to SETL2 before compiling. The custom conversion program written for this purpose does an excellent job of preserving indentation, vertical alignment, comments and other ergonomic aspects of the code. This has left open the option of converting to SETL2 as the primary development language.

main := func(args);

This is the main program for Delta. 'Args' is a tuple containing the parameters passed on the command line.

The iSETL interpreter is the 'main' program for an interactive session, but compiled code needs a preprogrammed sequence of commands. So far each experimenter has written a custom version of 'main' to carry out the desired tests. Some speculation has gone into the design of an interactive front-end for compiled Delta, either an interface to a source browser cum editor, or a SETL2 interpreter. While this issue is ultimately important in building the Delta user base, it has so far taken a back seat to the construction of basic transformation tools.

3.4 Version Control

The Delta program source resides in a production directory with a full audit trail, and represents a useable release of the Delta system. This serves as a backbone for independent development directories kept by several project programmers. The contents of a development directory are overlayed on the current production directory to produce experimental releases. In support of this, two scripts have been written (`idelta` and `cdelta`) to produce the composition of a developer's private directory with the public production directory and invoke either the iSETL interpreter or the SETL2 compiler on the result.

In addition special `checkout` and `checkin` scripts allow developers to move files between their directories and the production directory. These scripts check for potential conflicts between a known group of developers. They also maintain the audit trail and backup copies of recent work.

3.5 Cross Reference

A system is only "open" insofar as its components are easy to locate, understand and reuse. To enhance this quality of the Delta project, several tools have been added to the SETL programming environment. Together they provide an interactive cross-reference to the Delta project source files, fully integrated with the Gnu Emacs editor. The components of this system are:

A SETL editing mode teaches Emacs enough about the lexical structure of SETL to make its cursor motion, editing, and search commands recognize token boundaries.

A Tagsfile generator builds a catalogue of source locations where SETL identifiers are given new definitions. The tagsfile locates all statically recognizable definition sites using the various forms of declaration and side-effects in the iSETL dialect used in the Delta source code. A single tagsfile covers the whole Delta source. The file adheres to an Emacs format previously used for Fortran, C, yacc, and Eye.

The Gnu Emacs Tags functions have been enhanced to support structured code walk-through. While editing any part of the Delta source it is now easy to visit all definitions or uses of any function, variable, or field name. A stack of return locations is kept so that when the identifier's meaning has been sufficiently explored (or modified) the editing session can return to the spot from which it first departed. The return stack includes the remaining itinerary of any searches currently in progress. By directly modelling the call/return discipline this package supports code walk-through and makes it easy to validate changes to the code.

The Call Graph: One feature of an "open" system is the ability to replace lower level components to change the detailed behavior of higher level actions. Of course, the replacement must fill the needs of all its higher level callers. A **call graph** is a summary of component interrelationships that can be used to locate call sites of a given component. The tagsfile generator can also produce the call graph for a collection of iSETL modules. Module relationships can also be abstracted. Observing and quantifying the cross-module references has led to better choices about module boundaries in Delta.

4 Research Involvement

Subscript Classification (Paul Petersen)

One of the ways to improve data dependence information is to expand the applicability of the dependence tests to a larger percentage of the potential dependences. Classifying the sources of the unknown dependences is useful in determining where further effort may prove beneficial.

One experiment examined all of the unknown dependence arcs in a benchmark suite and categorizing them based on the type of coefficients of the loop indices. In each category the precedence was {Array, Variant, Invariant, Numeric}. If two or more different classification types were present in the same part of the subscript pair, the one with the higher precedence was chosen. Each group of coefficients was subdivided into four categories based on the types of their constant terms. By constant term we here mean any additive term not containing an index variable of some enclosing loop. The following functions were added to the Delta system:

classify_args_tree := func(pgm, ex, invar);

Return a string of 1 character labels which classify a subscript expression:

A=array	P=subroutine paramater
C=common variable	V=generic variable
F=function	X=unknown construct
I =invariant	0 =zero
N=numeric	1 =unity

sort_classify_set := func(res);
Return a sorted character string for the elements of set 'res'

classify_subscript := func(pgm, s1, exp1, s2, exp2, invar);
classify_subscript_pair := func(pgm, s1, exp1, s2, exp2, invar, ndirs);
Join the classification sets of each subscript and create a string tag to describe the dependence pair; record this.

partially_linear := func(pgm, s1, exp1, s2, exp2);
Determine if the dependence pair 'exp1', 'exp2' are partially linear with all coefficients of index variables constant.

	Coefficient Type			
Constant	Numeric	Invariant	Variant	Array
Numeric	15722	1079	6	499
Invariant	2908	4940	—	—
Variant	29492	3083	73	36
Array	25240	78	—	425

The above table summarizes this analysis. In each category we collected a weighted count of how many distinct dependence arcs could not be analyzed due to the lack of compiletime information characteristic of the category. The weighting factor equals the number of feasible directions of the potential dependence. The table ranks the most important sources of unknown dependence as:

1. variables that may be modified unpredictably during a loop,
2. subscripted subscripts,
3. loop invariants whose relation to the other terms in the subscript equations is not known.

Compilation techniques such as interprocedural analysis, and advanced induction variable recognition can help to reduce the first category. The problem with subscripted subscripts is more challenging and is usually resolved by the user asserting that the subscripting array is a permutation. Reducing the third category involves more complex analysis and propagation of known relationships between invariant variables.

Efficacy of Dependence Tests (Paul Petersen)

Despite the popularity of approximate data dependence tests, there has been little empirical analysis of their effectiveness. One research project[PP90] based on Delta analyzed some approximate tests including the GCD method and three variants of Banerjee's test.

- To evaluate the **accuracy** of these test, their outcomes were compared with an exact integer programming method.
- To evaluate their **effectiveness**, the Perfect Benchmark suite programs were processed, one subroutine at a time, through a Delta-based testbed system.

Two experiments were run using different sequences of dependence tests. Each potentially conflicting subscript pair was classified as described in 'Subscript Classification' above. The dependence test sequence applies only to subscript equations whose coefficients and constant terms are known at compiletime. A counter associated with each dependence test was incremented whenever that test was the first to detect independence. For Banerjee's tests and the integer programming test the increment was 1. The Constant, GCD, and integer programming test counters were incremented by the number of distinct direction vectors of the potential dependence since their results cover all dependences between the subscript pair. Counting in this way, the weight of each potential dependence grows exponentially with its level of nesting within DO loops.

Each of the two experiments consisted of two parts. The first part used the loop limits as they appear in the source program. In this case, the Banerjee Rectangular and Trapezoidal tests and the integer programming test do not apply to all loops since they require that the limits be known at compiletime. For the second part we assumed an arbitrary constant lower and upper loop limits and unit stride.

Did 'knowing' the loop bounds help much? The Banerjee Rectangular Test became more effective by 8.64%, but at the same time the Banerjee Infinity Test is reduced by 8.37% for a net gain of 0.27%. The reordering of the dependence tests between runs also illustrated that only 0.53% of the analyzable dependences needed to know the upper bounds of loops to resolve the equations. Bounds information may play a larger role with more advanced induction recognition, but does not by itself improve dependence testing.

Exact Integer Programming proved only 0.25% more accurate than the approximate tests across the whole benchmark. These results point to improving the quality of the information available at a potential dependence site as the most significant research goal in parallelizing FORTRAN.

Synchronization (Sam Midkiff)

The Delta system is being used as the implementation tool for two experiments concerning synchronization in shared memory multiprocessors. The first (partly implemented) experiment compares the effectiveness of several code generation techniques for optimizing synchronization instructions[Jay88, Li85, MP87]. Each of the optimization methods is being implemented in Delta. These 'synchronization minimizers' postprocess the result of a simple DOACROSS pass[Cyt86]. DOACROSS loops are partially parallel: they satisfy dependences between different iterations by synchronizing the parallel processors so that conflicting memory uses occur in their original serial order. Code has been prepared to insert POST/WAIT, and Alliant FX ADVANCE/AWAIT[All85] synchronization instructions into concurrent loops. Statistics will be collected on how much redundant synchronization is left after using each optimization method.

The second experiment is still in the planning stage. Its goal is to compare several synchronization technologies:

- Post and Wait,
- Advance and Await, and
- Process-based synchronization[SY88].

Using Delta, FORTRAN programs will be analyzed to determine how many bits of synchronization data are necessary to synchronize the programs using each of the methods. A compiler generated timer will be used to simulate the potential speedup of each synchronization method. This will let us study the tradeoffs between parallel speedup and the complexity of synchronization hardware.

Critical Path Length (Paul Petersen)

Using the same powerful notion of instrumenting a program in order to simulate an alternative model of its execution, an experiment is nearing completion to measure a program's execution time under ideal conditions. Like the MAXPAR simulation package[Che89], code is being added to serial programs so that when executed, an assessment can be made of their potential parallel execution times. So far the current work within Delta involved reproducing some of the MAXPAR results as cross-validation exercise. It is notable that the instrumentation approach made possible by Delta significantly lowers the time needed to acquire simulation data for a program.

The overall goal of the experiment is to characterize the performance of a particular run of a given program if only it could be compiled with perfect knowledge of the control and data dependences that arise during the run, and perfectly scheduled for execution by a parallel processor with no resource constraints. The metric we are most interested in initially is the operation count along the critical path of actual data and control dependences encountered in the run. By studying the wide gap between theoretically ideal and practically achievable compilation of actual application codes, this experiment can help set priorities and expose unforseen opportunities for optimization efforts.

Work in Progress

Several lines of development are now under consideration or already underway:

Data structures are being designed to represent the parallelized program, without losing the original sequential semantics needed for most analyses of the program.

The loop transformations on perfect loop nests should probably be extended and unified into a single step transformer along the lines suggested by [Ban90] and [WL90].

Modules will soon be needed to estimate the resources consumed by a transformed program if it were to execute on a given machine architecture. Specifically memory references should be counted and classified into different access patterns (e.g., vector write to private memory, or synchronized reduction to shared memory).

Demonstrations

The Delta Program Manipulation system was demonstrated outside CSRD

- at the Fall '90 DARPA contractors meeting in Chapel Hill, NC.
- the Supercomputing '90 Conference in New York City.

The audiences for these demonstrations were gently introduced to the whole topic of automatic program restructuring. On display were both the Delta project source code as seen through the interactive cross reference, and the interactive "try-it-and-see" interface. Loop Distribution, Interchange, Concurrentizing, Vectorizing and Stripmining were illustrated.

References

[All85] Alliant Computer Systems Corp., Acton, Massachusetts. *FX/Series architecture manual*, 1985.

[Amm90] Zahira Ammarguellat. A Control-Flow Normalizations Algorithm and its Complexity. CSRD Report 1042, Univ. of Illinois at Urbana-Champaign, Center for Supercomp. R&D, June 1990.

[Ban88] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[Ban90] Utpal Banerjee. Unimodular transformations of double loops. In *Proc. 3rd Workshop on Programming Languages and Compilers for Parallel Computing.* Pitman/MIT Press, August 1990.

[CFR$^+$88] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. Technical Report CS-88-16, IBM T.J. Watson Research Center, 1988.

[Che89] Ding-Kai Chen. MAXPAR: An execution driven simulator for studying parallel systems. MS thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomp. R&D, October 1989. CSRD Report 917.

[Cyt86] Ron Cytron. DOACROSS: Beyond Vectorization for Multiprocessors. In *Proc. 1986 International Conf. on Parallel Processing*, pages 836–844, August 1986.

[Eve] Shimon Even. *Graph Algorithms.*

[Jay88] Doddaballapur Narasimha-Murthy Jayasimha. *Communication and Synchronization in Parallel Computation.* PhD thesis, University of Illinois at Urbana-Champaign, 1988. CSRD Report 819.

[Li85] Zhiyuan Li. A technique for reducing data synchronization in multiprocessed loops. MS thesis, University of Illinois at Urbana-Champaign, Center for Supercomp. R&D, 1985. CSRD Report 521.

[Lov77] David B. Loveman. Program improvement by source-to-source transformation. *Journal of the ACM*, 24(1):121–145, January 1977.

[LT79] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[MP87] S.P. Midkiff and D. Padua. Compiler algorithms for synchronization. *IEEE Transactions on Computers*, C-36(12):1485–1495, December 1987.

[Pad89] David A. Padua. The Delta Program Manipulation system — Preliminary design. CSRD Report 808, University of Illinois at Urbana-Champaign, Center for Supercomp. R&D, June 1989.

[PP90] Paul Petersen and David Padua. Experimental evaluation of some data dependence tests. CSRD Report 1080, Univ. of Illinois at Urbana-Champaign, Center for Supercomp. R&D, December 1990.

[SM89] Harvey M. Salkin and Kamlesh Mathur. *Foundations of Integer Programming.* North-Holland, 1989.

[Sny90] W. Kirk Snyder. *The SETL2 Programming Language.* Courant Institute of Mathematical Sciences, May 1990.

[SY88] Hong-Men Su and P.C. Yew. Supporting data synchronizations on multiprocessor systems. CSRD Report 936, Univ. of Illinois at Urbana-Champaign, Center for Supercomp. R&D, August 1988.

[WL90] Michael E. Wolf and Monica S. Lam. Maximizing parallelism via loop transformations. In *Proc. 3rd Workshop on Programming Languages and Compilers for Parallel Computing.* Pitman/MIT Press, August 1990.

A The Delta System

The following is a collection of brief descriptions for the majority of the functions currently available in Delta. Every function which can be invoked from the top level of Delta has been included; this has tended to mix high level and low level functions. To relieve this, the functions are classified by topic and some discussion has been added outside the framework of the function-by-function documentation.

A.1 Fortran Programs as Data

The Scanner

The conversion FORTRAN77 → Delta internal form is carried out by a separate program called the 'scanner'. For each compilation unit in the FORTRAN source file, the scanner produces one SETL map. To Delta, this map *is* the program. Its hierarchical structure captures every significant semantic detail in the FORTRAN77 source code; allowing it to be accessed based upon its meaning, rather than its lexical structure.

run_scanner := func(name, output);
Run the FORTRAN 77 to DELTA conversion program. Supply the input and output file names to be used.

read_program := func(file_name);
Given a file name, return the Delta form of the first compilation unit found there.

For the remainder of this report and throughout the Delta source code, the variable name 'pgm' always refers to a program object of the type produced by 'read_program'.

The @expression Table

Delta represents FORTRAN expressions as labelled trees. The scanner produces a separate table (`@expressions`) containing the parse tree of each expression in the program. The trees are formed with explicit links (indices into `@expressions`). Each tree node is a map whose domain is a self-explanatory collection of field names, for example:

`{["op", "+"], ["args", [123,124]], ["type", "INTEGER"]}`.

In this example, the left argument to '+' is found at `@expression[pgm](123)`.

All expression nodes (except ',' operators) have a `@type` field for the data type of the operator's result. Currently the data frame size is kept only in the symbol table and not propagated throughout the expression tree. All expression nodes have an `@op` field. They may also have fields called **@name, @label, @value**, or **@args**. The following table describes how to interpret the `@op` field.

@op =		
INTEGER_CONSTANT	@value = integer	
REAL_CONSTANT	@value = 'real number'	
STRING_CONSTANT	@value = 'characters'	
LOGICAL_CONSTANT	@value = '.TRUE.' or '.FALSE.'	
HOLLERITH_CONSTANT	@value = 'H length characters'	
COMPLEX	@args = [real, imaginary]	
ARRAY_REF	@args = [array ID, subscript list]	
SUBSTRING	@args = [base variable reference, substring bounds]	
FUNCTION_CALL	@args = [function ID, parameter list]	
INTRINSIC_CALL	@args = [intrinsic ID, parameter list]	
RETURN*	@args = [label]	(in CALL parameter lists)
	omitted	(in ENTRY parameter lists)
LABEL	@label = 'statement name'	
IO*		(implied I/O unit or Format)
ID	@name = 'identifier'	
annotations:	@substituted = equivalent expression	
	@possible_values = {integers}	
	@value = integer	
U+ U- NOT	@args = [right]	
EQ NE LT LE GT GE	@args = [left, right]	
- / ** //	@args = [left, right]	
+ *	@args = [arg_1, arg_2, ... , arg_n]	
OR AND EQV NEQV	@args = [arg_1, arg_2, ... , arg_n]	
DO	@args = [iolist, iterator]	(for I/O implied DO loops only)
=	@args = [index ID, iteration space]	(for implied DO iterators only)
,	@args = [arg_1, arg_2, ... , arg_n]	(for parameter lists, subscript lists, io lists, implied DO iteration spaces, ...)

Explicit links from an expression to its subexpressions (`@args`) are very useful when a modification must affect a subexpression without worrying about where it actually occurs in the program (or how many occurrences may be sharing the subexpression structure).

repl_expr := func(pgm, e, old_value, new_value);
  Return a new copy of 'pgm' in which all occurrences in expression 'e' of the subexpression 'old_value' are (destructively) replaced by unique copies of subexpression 'new_value'. The substitution is not recursive. It affects `@substituted` expressions, but does not consider these when trying to match occurences of 'old_value'. It modifies expression 'e', without garbage collecting the subexpressions 'old_value', or copying 'e'.

replace_variable_uses := func(pgm,stmt,prototype_map);

find_and_replace := func(pgm,e,prototype_map);
  Return a new version of 'pgm' in which every ID node that is in the domain of 'prototype_map', and in expression 'e' (or in any expression of 'stmt') is replaced by a copy of its image in 'prototype_map'. Identity maps are allowed.

  This modification assumes there is *no structure sharing* and does not introduce any. IDs in `@substituted` expressions are not affected. The `@substituted`, `@possible_value`, and other annotation fields of replaced nodes are preserved.

copy_expression := func(pgm,e);
  Duplicate expression 'e'; return [new pgm, duplicate's index in `@expression`].

make_expr_node := func(pgm, node_contents);
  Return a new copy of 'pgm' and the index of a new expression therein which has the given 'node_contents'.

But, explicit links and two-part results are a nuisance when the expression must be reorganized, simplified, or repeatedly copied. For these actions, Delta expression transformers first 'implode' an expression, which makes a copy independent of the `@expression` table. The imploded form, called an **args_tree**, replaces the pointers of the `@args` field with the actual nodes to which they pointed. Args_trees are easier to manipulate because SETL automatically garbage collects unused nodes, and copies modified ones, and because they can be modified independent of any particular expression table. A reverse process ('exploding') embeds an entirely new copy of the expression tree within a given `@expression` table.

args_tree_implode_expression := func(pgm, e);
  Extract an expression from the `@expression` table of 'pgm'. Return a nested form of the expression that is independent of the expression table.

explode_args_tree := func(pgm, ex);
  Given a tree form expression, return a new copy of 'pgm' with the tree's components inserted into the expression table. Return [pgm, the index of the root table entry].

form_args_tree := func(op, type, left, right);
  Create a new 'tree' node.

SUB := func(left, right);

ADD := func(left, right);

MUL := func(left, right);

DIV := func(left, right);
Create a new arithmetic node in args_tree form.

COMMA := func(args);
Create a new ',' node for parameter lists, etc.

ARRAY := func(name, args);
Create a new ARRAY_REF node.

list_subscripts := func(pgm, ref);
Return a tuple of the subscript expression indices from an ARRAY_REF.

CST := func(val);
NEG := func(ex);
Create a constant node whose type is determined by the type of the argument.

result_type := func(op,left_type,right_type);
Find the resulting type from the given binary operation.

IS_CST := func(ex);
IS_ZERO := func(ex);
Determine if the expression tree node is a constant value.

COPY_ID := func(id_name, id_type);
MAKE_ID := func(pgm, id_name);
Create an args_tree ID node.

IFIX := func(child);
DYADIC_FN := func(fn, left, right);
ASSOC_FN := func(fn, left, right);
Create a new INTRINSIC_CALL tree node.

maximize_args_tuple := func(tuple);
Given a tuple of (non-negative) trees return the tree representation of the MAX over the list.

side_effect_free := func(pgm,e);
side_effect_free_args_tree := func(ex);
Return true if the given expression has no side-effects (or is OM).

invariant_args_tree := func(ex, invar);
Given a side_effect_free_args_tree 'ex', return true if all its free variables are in 'invar'.

equal_expressions := func(pgm, p, q);
equal_args_tree := func(p, q);
Are the two given expressions/trees structurally equivalent?

Algebraic simplification

simplify_args_tree := func(ex);
Return an arithmetically equivalent expression tree created by applying the following simplification transformations in the correct order.

coef_args_tree := func(ex, id);
Return a numeric coefficient of the variable 'id' in expression 'ex'. If the coefficient is not numeric then return OM.

all_variable_factors := func(ex);
Returns the set of variables that are contained in the formula of expression tree 'ex'.

contains_set_args_tree := func(ex, var_list);
Returns the subset of variables in 'var_list' that are contained in the top-level formula of expression tree 'ex'.

extract_args_tree := func(ex, var_list);
Given 'ex' (a term or sum of terms) and a list of variable names, decompose 'ex' into a mapping from the names onto either 0, or a term or sum of terms that mention the variable.

variable_factors := func(ex);
Returns the set of variables found as factors of 'ex'.

member_args_tree := func(ex, v);
Given an tree 'ex' which is either an identifier 'v' or a multiple thereof, return a subtree of 'ex' equal to 'v', else OM.

take_from_args_tree := func(ex, id);
Return the coefficients of the identifier 'id' from the multiplicative expression 'ex'.

combine_args_tree := func(ex);
Combine the coefficients of common terms and simplify.

dist_times_args_tree := func(args);
distribute_args_tree := func(ex);
Distribute '*' over '+'.

eval_func_args_tree := func(ex);
Return a simplified form of the INTRINSIC_CALL 'ex'.

flatten_args_tree := func(ex);
Flatten the tree structure around associative operators.

fold_args_tree := func(ex);
Combine constant arguments at every node. Use the `@substituted` annotations to replace ID nodes. Replace subtraction (`X-Y`) by the addition of the negation (`X+-1*Y`). Perform simple symbolic simplifications such as (`1*X`) $\rightarrow$ `X`, and (`0*X`) $\rightarrow$ `0`.

negate_args_tree := func(ex);
Given a tree that has recently been folded, multiply it by -1, in a way which leaves it as fully folded.

The `@statement` table

ASSIGNMENT	
	`@lhs`^{1}, `@rhs`^{1}
DO	`@follow`^{2}
	`@index`^{1}
	`@init_expr`^{1}, `@limit_expr`^{1}, `@step_expr`^{1}
IF, ELSEIF	`@expr`^{1}
	`@follow`^{2}
ENDDO, ENDTHEN	
	`@follow`^{2}
GOTO	`@target`^{2}
ARITHMETIC_IF, COMPUTED_GOTO, ASSIGNED_GOTO	
	`@expr`^{1}
	`@label_list`^{3} = [target$_1$^{2}, ...]
ASSIGN	`@lhs`^{1}
	`@target`^{2}
READ, WRITE, PRINT, OPEN, CLOSE, REWIND, BACKSPACE, ENDFILE, INQUIRE	
	`@s_control` = {['KEYWORD', expr^1], ...}
	`@io_list`^{1}
STOP, PAUSE	
	`@expr`$^{1\ 3}$
CALL, ENTRY	
	`@routine` = 'identifier'
	`@parameters`^{3}
RETURN	`@expr`$^{1\ 3}$
LABEL	`@label` = integer
ENDIF	

^{1} = index into expression table.
^{2} = 'statement name'.
^{3} omitted in some cases.

Table 1: Fields in the Abstract Syntax Tree for each Statement type

The executable statements of a program are collected in the `@statements` map. To describe the many characteristics of the different FORTRAN statement types, each statement is represented by a labelled map. The following subfields appear in every statement:

`@st`	= 'statement type'
`@next`	= 'name of the lexically next statement'

Table 1 lists all the FORTRAN77 statement types (`@st`) and the subfields used to capture their syntax and sematics.

Cross Reference, Flow Graph, Loop Nesting

One field provided by the scanner for some statement types helps Delta deduce the flow of control for compound constructs:

`@follow`	= name of the 'other statement' in a compound construct

For instance, DO/ENDDO statement pairs point to each other via the `@follow` field. For IF, ELSEIF and ELSE statements, `@follow` points to the matching ELSE, ELSEIF, or ENDIF. Each IF, ELSE and ELSEIF clause is ended by an ENDTHEN statement, whose `@follow` points to the ENDIF that terminates the whole mess. All one-line logical IFs are converted to IF/THEN/stmt/ENDTHEN/ENDIF sequences by the scanner. FORTRAN statement labels are attached to separate statements, of type 'LABEL'. They are just placeholders and branch targets.

All other fields are derived information, added by Delta during its program setup phase:

setup := func(pgm);
Return a copy of 'pgm' annotated with the derived fields.

`@in_refs`	= {indices of memory read expressions}
`@out_refs`	= {indices of memory write expressions }

Each `@expression` index in these sets is either an ID, an ARRAY_REF, or a SUBSTRING. `@in_refs` and `@out_refs` do not account for the 'hidden' side-effects of function and subroutine calls. These annotations are handled by the routines:

build_in_out_refs := func(pgm,s);

list_refs := func(pgm,s);
Add (list) `@in_refs` and `@out_refs` expression sets to statement 's'.

add_refs_to_program := func(pgm);
Return a copy of 'pgm' where all statements have valid `@in_refs` and `@out_refs` fields.

`@prev`	= 'name of the lexically previous statement'

Of course, the program counter does not always flow into a statement from its `@prev` statement, but so far every programmer who has written a low-level transformation has erroneously assumed this at least once. These bugs are fixed, but this misconcept about the `@prev` of a statement is subtle and insidious.

`@outer`	= 'statement name' of the innermost enclosing DO loop.

`@outer` is omitted for statements outside of loops. The `@outer` field of a DO statement names the next outer loop, not itself. The `@outer` field of an ENDDO names the matching DO.

`@successors`	= {'statement name's to which control may flow}
`@predecessors`	= {'statement name's from which control may arrive}

For example, most DO statements have two elements in `@successors`: the first statement in the body of the loop and the statement following the matching ENDDO.

flow_successors := func(pgm, stmt);
Return the set of statement names of 'pgm' to which control may flow from 'stmt'.

flow_predecessors := func(pgm, stmt);
Return the set of statement names of 'pgm' from which control may flow into 'stmt'. This requires o(`#statements[pgm]`) time to compute, so use the precomputed `@predecessors` field wherever possible.

A recurring issue in Delta is how much of the rich internal program representation must be rebuilt from scratch after each transformation. The flow graph is one example where coding complexity in the transformations must be traded off against the high cost of regenerating (via add_flow_graph) all the flow linkages in a program. The routine update_flow_info is one answer to the trade off. It works on a bounded section of the program provided that any explicit (i.e. non-default) links crossing the boundary are already correct. These conditions can be met by most transformations without sacrificing clarity or code space.

add_flow_graph := func (pgm);
Return a copy of 'pgm' where each statement has been annotated with: `@successors`, `@predecessors`, and `@prev` ignoring any existing values of those fields. The result also includes a `@final_statement` field that points to the lexically last statement.

update_flow_info := func (pgm, start_stmt, end_stmt);
Return a copy of 'pgm' where the `@predecessors`, `@successors`, and `@outer` fields have been updated over the given range of statements (connected by `@next`). Statements that are properly linked successors of statements in this range have their predecessor links updated to reflect changes within the range. The same is *not* true of properly linked predecessors, so beware of ranges that follow ELSEs and ENDIFs.

A.2 Analysis and Transformation

Data Dependence

Each pair of conflicting storage references is represented by a directed arc in a **data dependence graph** that indicates which reference is executed first in the original sequential program. The function `dependence_graph()` generates a complete graph for a nest of loops and saves the result in a program annotation (`@loop_info`). From this, `dependence_loop_info()` can derive a graph for any inner loop of the nest.

dependence_graph := func(pgm, doloop);
Build the dependence graph for the given loop. Return a set of tuples, whose elements are dependences. Each dependence has the following contents, in order:

1. source statement
2. sink statement
3. source atom expression number
4. sink atom expression number
5. variable causing the dependence
6. direction vector

7. dependence type ('f'=flow, 'a'=anti, 'o'=output)

derive_dependence := func(pgm, graph, s2);
Given a dependence 'graph' for loop s1 return the dependence graph for loop 's2' which is a subloop of s1.

dependence_loop_info := func(pgm, s1);
Annotate the `@loop_info` database with the dependence graph for loop 's1'. Compute the graph from an enclosing loop or compute it directly for the outermost loop.

print_graph := func(pgm, l, var_id);
Print out the dependence graph for a loop nicely. Uses the dependence graph in `@loop_info`. If var_id is given, only print dependences for that variable.

dependences := func(pgm, s1, s2, doloop);
Build the dependences between the two given statements.

intersect := func(pgm, s1, s3_refs, s2, s2_refs, dirs, dep_type);
Intersect the two given reference lists and return a dependence for each intersection and each element of the set of directions given.

plausible_directions := func(pgm, s1, s2, doloop);
Return a set of plausible direction vectors for statements S1 and S2. Each direction vector is a tuple with the first element being the outermost common loop, and each element being a string containing one or more of the characters '=', '¡', or '¿'.

ignore_non_equal := func(graph, ignore);
Ignore a set of variables for which non-'=' directions should be excluded from the graph, e.g. localizable scalars.

The computation-intensive part of data dependence analysis decides whether a system of equations (the equated subscripts of two array accesses) has an integer solution in a given region of Z^n (the iteration space of the surrounding loops). Integer Programming techniques[SM89] can answer this question accurately. However, faster approximate techniques[Ban88] are thought to be more practical. An approximate test will sometimes predict a non-existant solution to the system of equations, but conservative use of the test results never leads to incorrect code, just to missed opportunities for optimization.

same_test := func(pgm, s1, e1, s2, e2, dirs, dep_type);

dd_tests := func(pgm, s1, exp1, s2, exp2, dirs, proto);
This routine is called only if the lexical names of ref1 and ref2 are identical. It returns false if it can be shown that a dependence does not exist, or true otherwise.

nonlinear_test := func(pgm, s1, exp1, s2, exp2, dirs);
This test counts the number of non-linear potential dependences.

dd_tree_tests := func(pgm, s1, exp1, s2, exp2, dir, proto);
Expand the direction-vector tree until a true/false result is found. If the result at the current node is unspecified then recurse down.

dd_simple_tests := func(pgm, s1, exp1, s2, exp2, dirs, dir_count);
constant_dd_test := func(pgm, s1, exp1, s2, exp2, dirs);
gcd_dd_test := func(pgm, s1, exp1, s2, exp2, dirs);
Examine the `@simple_dd_tests` options and invoke each routine in sequence until one of the tests either returns true or false.

dd_complex_tests := func(pgm, s1, exp1, s2, exp2, dir, dir_count);
infinity_dd_test := func(pgm, s1, exp1, s2, exp2, dir);
banerjee_dd_test := func(pgm, s1, exp1, s2, exp2, dir);
trapazoid_dd_test := func(pgm, s1, exp1, s2, exp2, dir);
int_prog_dd_test := func(pgm, s1, exp1, s2, exp2, dir);
display_dd_result := func(name, exp1, exp2, dir);
Examine the `@complex_dd_tests` options and invoke each routine in sequence until one of the tests either returns true or false.

jayasimha_test := func(pgm, s1, exp1, s2, exp2, dirs);
This test is to locate subscript pairs in which all coefficients are integral, but some are different.

exact_subscript_test := func(pgm, s1, exp1, s2, exp2, directions);
Invoke the exact linear programming dependence test

trapazoid_direction_test := func(pgm, s1, ex1, s2, ex2, dirs);
trapazoid_function_bounds := func(pgm, s1, ex1);
Find the function bounds for expression 'ex1' at statements 's1'. The value OM is returned upon error.

banerjee_trapazoid := func(pgm, s1, ex1, s2, ex2);
Determine the function bounds for two expressions 'ex1' and 'ex2' in statements 's1' and 's2'. If the constant term noes not lie between the computed bound then no dependence is possible.

banerjee_quadrant := func(pgm, s1, ex1, s2, ex2, directions);
Banerjee_quadrant is called if both subscript expressions are linear functions of induction variables. This routine will only work properly when all coeficients are integers, and the loop has been normalized.

infinity_test := func(pgm, s1, ex1, s2, ex2, directions);
The infinity_test is called if both subscript expressions are linear functions of induction variables. This routine will only work properly when all coeficients are integers, and the step on the loops are all positive.

banerjee_inequality := func(pgm, s1, ex1, s2, ex2, directions);
Banerjee_inequality is called if both subscript expressions are linear functions of induction variables. This routine will only work properly when all coeficients are integers, and all loop limits are integers. It also assumes that the loop has been normalized and has a step of 1.

In posing the integer programming problem, unknowns which are invariant in some surrounding loops are symbolically cancelled. The function `cancel_common_terms(ex1, ex2, vars)` eliminates variables that are additive in both subscript expressions (`ex1, ex2`) and also belong to the invariant set (`vars`) of a given loop. The particular loop to use is the outermost one in which the conflicting references may occur in different iterations.

select_invariant := func(pgm, common, dir);

select_invariant_dirs := func(pgm, common, dirs);
Select the set of invariant variables based on the current direction vector. Given a set of direction vectors, return the most restrictive set of invariant variables.

cancel_common_terms := func(ex1, ex2, invar);
Given two expressions 'ex1' and 'ex2', return a tuple of the expressions with all common additive terms in 'invar' canceled.

unknown_test := func(pgm, s1, exp1, s2, exp2, dirs, dir_count);
Test the subscript pair to see if any direction vector in 'dirs' supports a large enough invariant set to break an 'unknown' test result.

symbolic_lower_bound := func(pgm, s, id, indx);

symbolic_upper_bound := func(pgm, s, id, indx);
Given DOloop 's' in 'pgm' and the indices 'indx' of the enclosing DOloops, return a map representing a linear function of indx that computes either the lower/upper bound of s, or, if 'id' is set, id – bound. If the function is nonlinear and the `@loop_info` contains a guess of s's bound, use the guess in place of the actual bound.

A.3 Statement Manipulation

list_stmts := func(pgm);
Return a list of all of the statements in the program

stmts_of_type := func(pgm,st);
Return a tuple (ordered by the `@next` links) of all statements of the given types; 'st' can be either a string or a set of strings.

reorder_statements := func(pgm,start_block, end_block, new_order);
Given a range of statements and a new order, rearrange the `@next` and `@prev` links to put them in that order. Don't update the flow graph or `@outer` fields.

connect_two_statements := func(pgm,first_stmt,second_stmt);
Set the `@next` and `@prev` fields of two statements to point to each other.

delete_stmt := func(pgm,s);
Disconnect a statement from the lexical and flow graphs of the program. The statement remains in the statement table; its (invalid) links are unchanged. This is not sufficient to delete pieces of a compound construct (DO, IF, GOTO/LABEL, ...)

add_after_stmt := func(pgm, preceding_stmt, new_stmt);
Insert a statement after another one.

make_assignment_stmt := func(pgm, p, q);
Create an assignment statement 'p = q'. Don't link it into the program.

Procedure CALLs

The following utilities are concerned with CALL statements and the various forms of entry points into FORTRAN compilation units. The most significant capability of Delta with respect to interprocedural analysis is its inline expansion of subroutine calls. Given a program block, a subroutine, and a statement in the program block that calls the subroutine, the inline expander replaces the call statement with the body of the called subroutine. It changes the variable names used by the subroutine, so that the expanded program block is functionally equivalent to the origrinal program block. This facility is expected to prove useful in parallelizing loops where the presence of a subroutine call inhibits parallelization. At a higher level of abstraction, it eliminates some of the need for interprocedural analysis by destroying subroutine calls.

make_call_stmt := func(pgm, name, p);
Create a subroutine call statement, where p is a ',' node for the parameters. Don't link it into the program.

call_stmts := func(pgm);
Builds a map from routine names onto non-empty sets of statements where they are CALLed.

function_calls := func(pgm);
Find all function calls and return a mapping pointing to the statements wherein they occur.

inline_expand := func(pgm1, call_stmt, pgm2, called_routine);
Expand the subroutine CALL at call_stmt in pgm1 using the body of the called_routine (a named entry point) from pgm2.

routine_name := func(pgm);
Find the routine name of the lexically first ENTRY point, if any if none, return OM. Let the caller convert it to 'MAIN' or whatever!

entry_points := func(pgm);
Find all ENTRY points of a given program. Return a map from routine name to ENTRY statement tag for that name.

float_entry_points := func(pgm);
Move all ENTRY statements to the top of the routine. Follow each one with a GOTO/LABEL pair to its original location. This produces a unique empty section of code following each entry point. Lexical ordering of ENTRYs is preserved, therefore the routine name is not changed.

move_ENTRY_after := func(pgm, dest, entry);
Move a statement of type ENTRY to after dest.

Branching and Conditionals

make_goto_stmt := func(pgm, st);
Create an UNCONDITIONAL_GOTO statement that branches to statement st. Don't link it into the program.

make_label_stmt := func(pgm, lb);
Create a LABEL statement with label lb. Don't link it into the program.

numeric_labels_used_in := func(pgm);
Find all numeric labels used in pgm.

delete_trivial_gotos := func (pgm);
Remove UNCONDITIONAL_GOTOs that go to the next statement. Remove the LABEL there too if possible.

remove_gotos := func(pgm);
Change any eligible IF-GOTO statements in the program to block IFs. Currently using an ad hoc approach; future home of full flow normalization[Amm90].

convert_arithmetic_if := func(pgm,stmts,s);
Try to convert an arithmetic IF into an IF-GOTO-ENDTHEN-ENDIF. This is possible if two of the target labels are the same, and if one of the targets is the following statement. Stmts is the `@statements` section of pgm separated out for convenience. Return the modified program and statements and the FIRST statement in the resulting sequence of statements after converting the IF if successful. Otherwise, just return the arguments unchanged.

invert_if_condition := func(pgm,if_stmt);
Invert the condition of the IF statement in the indicated statement.

split_ELSEIF := func (pgm, old_elseif);
Convert an ELSE IF clause into the ELSE clause of a new outer IF statement.

delete_IF := func(pgm, IF_stmt);
Remove an IF/ELSE/ENDIF structure. Connect the THEN and ELSE clauses as in-line code. Caller is responsible for preserving program semantics!

DO Loops

list_do_loops := func(pgm);
Return a tuple of [do-start, do-end] pairs

display_do_loops := func(pgm,extra_stuff);
Write out the DO and ENDDO statements in the program

list_loop_body := func(pgm,do_stmt);
Return a tuple listing the statements in the given DO loop

enclosing_loops := func(pgm,s);
Build a tuple giving the loops enclosing the given statement, with the outermost loop first and innermost last. Return an empty tuple if there are no loops around this statement.

common_loops := func(pgm,s1,s2);
Return a tuple of all DO loops that enclose both statements

loop_index := func(pgm,l);
Return the name of the index variable for the given loop.

delete_loop := func(pgm,do_stmt);
Remove a DO loop from around a set of statements. Reconnect the lexical and flow graphs of the program. The DO and ENDDO statements are deleted from the `@statement` and `@loop_info` tables. The `@outer` fields are updated.

make_do_loop := func (pgm, after, index, init, limit, step, before);
Make a new DO/ENDDO statement pair. Put the DO after 'after', giving it the index variable 'index' and bounds formed from the args_trees 'init', 'limit', and 'step'. Put the ENDDO before 'before'. Statements between 'after' and 'before' (if any) form the body of the loop. Their `@outer` field is updated, but their `@loop_info` is not (yet) affected.

initialize_local_variable := func(pgm, s, v, expr);
Create a statement or loop to initialize a local variable

A.4 Preconditioning

Control Flow

control_flow_graph := func(pgm, stmt_list);
Return a control flow graph for the list of statements contained in stmt_list. The value is a mapping of statements onto a set of arcs.

dominates := func(idom, s1, s2);
Return true iff 's1' dominates 's2'.

all_immediate_dominators := func(pgm);
all_immediate_postdominators := func(pgm);
Returns a mapping from every statement in the program to its immediate (post)dominator statement.

loop_immediate_dominators := func(pgm, doloop);
loop_immediate_postdominators := func(pgm, doloop);
Returns a mapping from every statement in doloop to its immediate (post)dominator statement.

immediate_dominators := func(stmt_list, CFG, R_CFG);

immediate_postdominators := func(stmt_list, CFG, R_CFG);

Returns a mapping from statements in 'stmt_list' to their immediate dominator statement: the closest statement that appears on every path from the entry to the statement. Tarjan's flow dominator algorithm is used [LT79].

In the map, idom(first stmt_list) = first stmt_list is added to make the resulting map everywhere defined.

dominance_frontier := func(idom, root, CFG);

control_dependence := func(pgm, stmt_list);

Given an immediate dominator map and the 'root' of its corresponding control flow graph, (or a 'pgm' from which a *post*dominator map can be generated for a particular 'stmt_list') this returns a map from each statement X to all statements Y such that X (post)dominates a (successor) predecessor of Y but does not strictly (post)dominate Y. This algorithm forms part of PTRAN's Static Single Assignment Form conversion described in [CFR+88].

invert_graph := func(graph);

Invert a graph represented as a mapping of items onto sets of items.

strongly_connected_components := func(stmts,graph,l);

Given a set of statements and a graph connecting them, return the strongly-connected components as a set of disjoint subsets of 'stmts'. For now 'graph' is represented as a set of ordered pairs representing the directed edges between statements. Tarjan's algorithm [Eve] is used.

build_pi_blocks := func(pgm, l);

Build the strongly-connected components (aka pi-blocks) for loop 'l'.

topologically_sort := func(nodes,edges);

Given node and edge sets of a graph, return a tuple of nodes in an arbitrary total order which satisfies the partial ordering induced by the edge relation. The edge set must not have any cycles involving members of 'nodes'.

build_all_prior_sets := func(pgm);

Add prior sets as `@prior` to each DO statement in the program

new_prior := func(pgm, s1, s2, l);

Do the prior test for two statements with respect to a loop. i.e. return true if there is a flow path from the first statement to the second within a single iteration of the loop. Assume that the prior sets are attached to the loop header.

Reducing irrelevance

dead_code_elimination := func(pgm);

Eliminate dead (i.e. unused) assignment statements in 'pgm'. Remove IFs and DOs that become empty along the way.

delete_ELSEIF := func(pgm,ELSEIF_stmt);
Deadcode removal of ENDTHEN / ELSEIF pairs. Assumes ELSEIF does not control any meaningful statements. Does not remove any ELSE parts of the ELSEIF construction. Semantically too wierd for any use besides deadcode elimination.

unreachable_code_elimination := func(pgm);
Delete statements that cannot be reached from the `@initial_statement` via `@successor` links and that are not needed for the lexical integrity of the program. Also delete statements disconnected from the lexical graph. Assumes that the reachable set is a subset of the statements connected by `@next` and `@prev` links in the lexical graph. Assumes (ANSI F77) that there are no branches into DO loops, or into IF, ELSE, or ELSEIF blocks.

Induction

induction_variables := func(pgm, doloop);
Find induction variables in a loop. Replace uses of the variable with the equivalent value expressed as a tuple of expressions, where the first element is the constant term, the second the increment due to the inner loop, the next the increment due to the next inner loop, etc.

test_for_increment := func(pgm,s,loop_invariants);
See if the given statement is an increment. If it is, return a tuple with the the lhs variable name and the set of rhs terms other than the lhs variable. Return OM if not an increment.

not_always_executed := func(pgm, stmt, loop_body);
Return TRUE if the given statement isn't executed exactly once every time through the given loop.

substitute_induction_variables := func(pgm, l);
Given a set of induction variable assignments for a loop, substitute uses of the variables in loop l with the equivalent value in terms of the (normalized) loop index variable. Return the modified program and a count of variables removed.

substitute_all_induction_variables := func(pgm);
Call substitute_induction_variables on each loop of the program.

induction_loop_info := func(pgm, s1);
Annotate the @loop_info database with the induction variables for loop 's1'.

Invariance

invariant_expression := func(pgm, e, invariants);
Return true if all of the names that appear in the given expression are in the given set. The expression is then invariant iff all the function calls and intrinsic calls are deterministic (checks for this not included).

loop_invariant_variables := func(pgm,l);
Return a set of variable names that are not modified in the given loop

invariant_loop_info := func(pgm, s1);
Annotate the @loop_info database with the invariant variables for loop 's1'.

Canonical iteration space

normalize_loop := func(pgm, doloop);
To normalize a loop, subtract the lower-bound from the upper and lower bounds, rename the loop index variable to a new name, and subst all of the occurences of the loop index variable by the new formula. Locate all exits from the loop, and restore the index value.

rename_do_loop_index := func(pgm, doloop, index, repl);
Find all exit paths from loop 'doloop' and make sure that the line index = new_value is added to each.

normalize_all_loops := func(pgm);
Normalize all of the loops in the program

Forward substitution

propagate_constants := func(pgm);

ok_to_substitute := func(pgm, e, OK_names);
Do forward substitution of scalar variables throughout pgm. Attach a subtree to each substitutable expression giving either an equivalent expression tree (in @substituted) or a set of possible constant values (in @possible_values). Expressions are substituted only if they are free of function calls and array references.

evaluate_scalar_assignment := func(pgm,expr, stmt_IN, DEF_values);
Try to evaluate the expression given the scalar definitions that reach it. If a non-constant value reaches any of the rhs exprs, return OM, otherwise return the set of possible values to be attached to the statement. Return an empty set if evaluation will always be impossible. DEF_values is a mapping of DEFs to the set of values for that def.

Alternative: use a simple evaluator that takes a binding environment as parameter, collect possible values from the evaluation in all possible binding combinations for the expression's input variables.

evaluate_integer_expression := func(pgm,e);
Try to evaluate an expression to an integer value. Return OM if it is impossible.

clean_expressions := func(pgm);
Remove the propagated constants and expressions from pgm.

Scalar expansion

localizable_scalars := func(pgm,l);
Compute the set of scalars that could be localized to the given loop. Assumes that in_refs and out_refs are correct.

localizable_loop_info := func(pgm, s1);
Annotate the `@loop_info` database with the localizable variables for loop 's1'.

expand_scalars := func(pgm,names,enclosing_loops);
Expand each of a set of scalars into arrays. Add one dimension for each loop in the enclosing_loops tuple (which is ordered with the outermost loop to be expanded first). Dimension it as `(*,*, ...,*)` (fill in the bounds later). Assume that the loops are normalized, and use the inner loop as the leftmost subscript.

A.5 Restructuring

Loop interchange

permutation_to_swaps := func(permutation);
Given a tuple representing a permutation, e.g. `[3,1,2]`, return a tuple of ordered pairs giving exchanges of *adjacent* elements to make in order to change `[1,2,3]` into that permutation. For `[3,1,2]`, one possible sequence of exchanges is `[[2,3], [1,2]]`. The strategy taken is to swap the element that goes to the last position into place (i.e. 3 in the example is swapped with 2, then 2 is swapped with 1 to bring it to its final position), then repeat as necessary for the second-to-last, ...Note that this is $o(n^2)$, like a bubble sort (which it strongly resembles).

permute_loops := func(pgm, old_order, new_order);
Given an old outside-in order for a set of perfectly nested loops and a new order, return a copy of 'pgm' in which the loop nest is permuted into the new order.

interchange_loops := func(pgm,s1,s2);
Return a copy of 'pgm' in which the perfectly-nested loops 's1' and 's2' are interchanged.

legal_to_interchange := func(pgm,outer_do,inner_do);
Test whether interchanging the given loops would violate a data or control dependence.

interchange := func(pgm, l1, l2);
Cover function for interchange_loops that first checks to see if it is legal to interchange the loops.

Loop distribution

distribute_loop := func(pgm,l,break_after);
Break a loop 'l' in two after one of its statements 'break_after'.

smash_loop := func(pgm,doloop);
Build the piblocks for a loop, then use them to smash the loops into tiny pieces. Start by smashing the inner loops recursively (therefore this is inside-out distribution). Return the modified program and the last resulting ENDDO.

Stripmining[Lov77]

strip_vertical_inner := func(pgm, DO_loop, stripsize);
strip_vertical_outer := func(pgm, DO_loop, stripsize);
strip_horizontal_inner := func(pgm, DO_loop, stripsize);
strip_horizontal_outer := func(pgm, DO_loop, stripsize);
stripmine := func(pgm, DO_loop, length);

Return a version of 'pgm' and the statement tags of two perfectly nested loops which together control the iteration space of the given 'DO_loop'. The iteration space is split into strips of a positive integer size computable by the expression 'stripsize' (or given by the constant 'length'). The bounds expressions of 'DO_loop' should be invariant in the loop.

This technique is in widespread use by vectorizers, concurrentizers, and ordinary compilers that attempt to improve data locality. For example in strip_vertical_inner, the inner loop carries the original loop's index variable over 'strips' of a given or computable size using the same stride as the original loop. The new outer loop schedules enough strips to cover the original iteration space.

Vectorizing and Concurrentizing

tiny_vectorizer := func(pgm);

Return a version of 'pgm' in which loops are marked as DOALL or SUM based on a small amount of dependence pattern recognition.

trivial_subscript_test := func(pgm);

Look at all of the dependences in `@loop_info[pgm]`. Eliminate any which have: A '¡' in some direction vector position. A source and sink subscripted by the index for the corresponding loop. Return the reduced dependence graph.

trivially_parallel := func(pgm,l);

Return true if there are no non-'=' dependence arcs in the graph for this loop.

trivial_summation := func(pgm,l);

See if this loop is a sum of trivial form, e.g. scalar = scalar + 'invariant stuff'

Instrumentation

instrument_all_outer_loops := func(pgm);

Return a copy of 'pgm' in which each outermost loop is bracketted by a pair of subroutine calls that can collect and identify timings of individual loop nests.

instrument_do_loop := func(pgm, doloop);

Return a copy of 'pgm' in which the 'doloop' is bracketted by calls to DO$ENTRY and DO$EXIT with the statement tag of the loop as their argument.

critical_path_program := func(pgm);

Return a version of 'pgm' instrumented so that executing it serially will measure the longest critical path of data or control dependence for that program run.

Output

write_program := func(pgm);

display_program := func(pgm);

write_program_to_file := func(pgm, fname);

write_program_fd := func(pgm, all_info, fd);

Write out a program unit with the given name.

write_stmt_string := func(string, indent, heading_width, fd);

Write out a string as a statement with indentation level 'indent' and with continuation lines preceded by 'heading_width' spaces. Analyze while printing to see if continuation lines can be indented safely. Indent level should not include the initial 6 columns. also, the initial line will not have indentation or headings printed by this program; that is the responsibility of the caller.

type_decls := func(pgm, fd);

parameter_decls := func(pgm, fd);

format_dim := func(pgm, dim);

array_decls := func(pgm, fd);

common_decls := func(pgm, fd);

equivalence_decls := func(pgm, fd);

data_decls := func(pgm, fd);

save_decls := func(pgm, fd);

external_decls := func(pgm, fd);

Produce the declarative statements that establish the evironment for 'pgm's execution. FORTRAN77 output syntax only!

write_stmt := func(pgm, stmt, all_info, marking);

write_stmt_fd := func(pgm, stmt, all_info, marking, fd);

Write out executable statement 'stmt'. Return the next statement, or OM if none. Print extra stuff if 'all_info' is set, prepend 'marking'.

write_all_stmts := func(pgm, initial, final, all_info, fd);

Write out all the statements in the given set that are in the chain starting with 'initial', connected by `@next`, and ending with 'final'.

write_do_loop := func(pgm, do_stmt);

write_do_loop_fd := func(pgm, do_stmt, fd);

Write out one do loop.

statement_label_value := func(pgm,s);

Return a string form of the numeric statement label for statement 's'.

write_format_stmts := func(pgm, fd);

Write out any format statements in the program.

format_expr := func(pgm, e, give_values);

write_expr := func(pgm, e, give_values);

writeln_expr := func(pgm, e, give_values);

Convert an expression tree into a string for output. Print out an expression tree. Write out an expression labelled by its number followed by a newline.

new_identifier := func(pgm, id);

create_identifier := func(pgm, base_id, id_type);

create_simple_variable := func(pgm, name, type, size);

new_critical_identifier := func(pgm, id);

make_label := func(prefix);

Serve different needs for the production of new names for compiler generated components of the program.

A.6 SETL Utilities

integer_to_string := func(n);

Return the character string representation of the integer 'n'.

string_to_integer := func(s);

Return the character string representation of the integer 'n'.

misc_to_string := func(int_or_string);

Given a parameter that is an integer or a string, return the string equivalent.

pad_string := func(s,length);

left_pad_string := func(s,length);

Given a string and a minimum length, pad it with blanks to that length if necessary.

compact_object := func(obj);

Returns its argument with structure sharing among leaf nodes.

negp := func(x);

posp := func(x);

Return the negative/positive part of an integer.

iNEG := func(a);

iADD := func(a, b);

iSUB := func(a, b);

iMUL := func(a, b);

Operate on the arguments taking care of infinities and OM.

gcd := func(a, b);

gcdn := func(list);

Compute the GCD of a pair (list) of integers.

divides := func(a, b);

Determine if a divides b.

between := func(a, b, c);
Determine if the relation [a ¡= b ¡= c] is true.

tuple_reverse := func(x);
Reverse the order of elements in a tuple.

tuple_index := func(x, T);
Return the index of the first occurence of 'x' in tuple 'T', or OM if none.

laminate := func(list1, list2);
Laminate two tuples into a tuple of tuples.

split_tuple := func(tpl, test);
Split a tuple into two parts, one which passes the test and the other not.

list_strings := func(separator, strings);
Concatenate a set or tuple of strings, separate them with 'separator'.

commas_between := func(list);
Concatenate strings in list, separating them by ', '.

limit_tuple_length := func(list, limit);
Given a tuple of strings return a tuple of tuples which are all less than the limit when separated by commas.